

Федеральное агентство по образованию

Рязанский государственный радиотехнический университет

А.В. МАРКИН

# **ПОСТРОЕНИЕ ЗАПРОСОВ И ПРОГРАММИРОВАНИЕ НА SQL**

Учебное пособие

**Допущено Учебно-методическим объединением вузов по университетскому политехническому образованию в качестве учебного пособия для студентов высших учебных заведений, обучающихся по специальности 230201 «Информационные системы и технологии»**

Рязань 2008

УДК 004.655  
ББК 32.973.26-018.2

*Рецензенты:*

кафедра Информационных технологий Рязанского института (филиала)  
Московского государственного открытого университета (заведующий кафедрой  
канд. техн. наук, доцент Занин А.Е.);

канд. техн. наук, профессор, заведующий кафедрой Информационных  
технологий Сочинского государственного университета туризма и курортного  
дела Дрейзис Ю.И.;

канд. техн. наук, доцент кафедры Систем автоматизированного проектирования  
вычислительных средств Рязанского государственного радиотехнического  
университета Таганов А.И.

Построение запросов и программирование на SQL: учеб. пособие /  
А.В.Маркин. – Рязань: РГРТУ, 2008. – 312 с.

Подробно рассмотрены все основные синтаксические конструкции,  
применяемые при построении запросов и программировании на языке  
структурированных запросов (SQL).

Весь теоретический материал пособия в полной мере проиллюстрирован  
множеством примеров запросов и программ на учебной базе данных, являющейся  
упрощенной моделью базы данных реальной информационной системы.

Разработанный автором Internet-ресурс позволяет обучаемым как проверить  
полученные теоретические знания языка SQL СУБД Firebird, так и приобрести  
практические навыки построения запросов и программирования, выполнив  
предлагаемый лабораторный практикум по всем разделам учебного пособия.

Для студентов вузов, обучающихся по специальностям, связанных с  
разработкой, эксплуатацией и сопровождением баз данных. Может быть полезным  
всем изучающим SQL самостоятельно.

*База данных, SQL, запрос, клиент-сервер, хранимая процедура, триггер,  
транзакция*

Печатается по решению редакционно-издательского совета РГРТУ

ISBN 978-5-7722-0285-2

© Рязанский государственный  
радиотехнический университет, 2008

## Оглавление

Предисловие.....	6
Введение.....	6
<b>1. Реляционная модель данных</b> .....	<b>7</b>
1.1. Определение реляционной модели.....	8
1.2. Таблицы.....	11
1.3. Первичные ключи.....	13
1.4. Внешние ключи.....	14
1.5. Связи между таблицами.....	15
1.6. Нормализация отношений.....	17
1.7. Модели базы данных на логическом и физическом уровнях.....	19
1.8. Целостность данных.....	22
1.9. Архитектура "клиент-сервер".....	23
Контрольные вопросы.....	32
<b>2. Введение в SQL</b> .....	<b>32</b>
2.1. Объекты структуры базы данных.....	33
2.2. Функции SQL.....	36
2.3. Достоинства SQL.....	40
2.4. SQL-сервер Firebird.....	43
2.5. Правила синтаксиса и основные запросы SQL.....	47
2.6. Формы использования SQL.....	52
2.7. Имена объектов в SQL. Константы, отсутствующие данные.....	53
2.8. Выражения.....	55
2.9. Типы данных.....	57
Контрольные вопросы.....	61
<b>3. Язык выборки данных</b> .....	<b>61</b>
3.1. Синтаксис запроса SELECT.....	61
3.2. Запросы к одной таблице.....	63
3.2.1. Предложения SELECT и FROM.....	65
3.2.2. Предложение WHERE.....	70
3.2.2.1. Простое сравнение.....	71
3.2.2.2. Проверка на принадлежность диапазону значений.....	72
3.2.2.3. Проверка на соответствие шаблону.....	73
3.2.2.4. Проверка на наличие последовательности символов.....	74
3.2.2.5. Проверка на совпадение с началом строки.....	75
3.2.2.6. Проверка на членство в множестве.....	75
3.2.2.7. Проверка значения на NULL.....	76
3.2.2.8. Проверка двух значений на отличие.....	77
3.2.2.9. Составные условия поиска.....	77
3.2.3. Функции в SQL.....	79
3.2.3.1. Классификация функций.....	79
3.2.3.2. Скалярные функции.....	81
3.2.3.2.1. Строковые функции.....	81
3.2.3.2.2. Числовые функции.....	87
3.2.3.2.3. Функции даты и времени.....	89
3.2.3.2.4. Функция преобразования типа.....	92
3.2.3.3. Агрегатные функции.....	94
3.2.3.3.1. Общее описание агрегатных функций.....	94
3.2.3.3.2. Вычисление среднего значения.....	94
3.2.3.3.3. Вычисление суммы значений в столбце.....	95
3.2.3.3.4. Вычисление экстремумов.....	96

3.2.3.3.5. Вычисление количества значений в столбце .....	96
3.2.3.4. Функции на списке значений .....	97
3.2.3.4.1. Функции MAXVALUE и MINVALUE .....	97
3.2.3.4.2. Функция LIST .....	98
3.2.3.5. Дополнительные возможности вывода в предложении SELECT .....	98
3.2.3.5.1. Операция выбора CASE .....	99
3.2.3.5.2. Функция COALESCE .....	101
3.2.3.5.3. Функция NULLIF .....	102
3.2.3.5.4. Функция IIF .....	103
3.2.3.5.5. Функция DECODE .....	104
3.2.4. Сортировка результатов запроса .....	105
3.2.5. Предложение GROUP BY .....	109
3.2.6. Предложение HAVING .....	113
3.3. Многотабличные и вложенные запросы .....	116
3.3.1. Соединения таблиц .....	117
3.3.1.1. Неявное соединение таблиц .....	117
3.3.1.2. Явное соединение таблиц .....	121
3.3.1.3. Стандартные соединения (объединения) таблиц .....	125
3.3.1.3.1. Декартово произведение .....	125
3.3.1.3.2. Эквисоединение .....	126
3.3.1.3.3. Естественное соединение таблиц .....	127
3.3.1.3.4. Композиция .....	127
3.3.1.3.5. Тета-соединение .....	128
3.3.1.4. Соединение таблицы со своей копией .....	129
3.3.2. Запросы с вложенными запросами .....	130
3.3.2.1. Виды вложенных запросов .....	130
3.3.2.2. Запросы с простыми подзапросами .....	132
3.3.2.2.1. Простые подзапросы в предложении WITH .....	132
3.3.2.2.2. Простые подзапросы в предложении SELECT .....	136
3.3.2.2.3. Простые подзапросы в предложении FROM .....	137
3.3.2.2.4. Простые подзапросы в предложениях WHERE и HAVING .....	140
3.3.2.3. Запросы со связанными подзапросами .....	145
3.3.2.3.1. Связанные подзапросы в предложении SELECT .....	146
3.3.2.3.2. Связанные подзапросы в предложениях WHERE и HAVING .....	148
3.3.2.4. Предикаты ANY и ALL .....	153
3.3.2.5. Предикат SINGULAR .....	156
3.3.2.6. Предикат EXISTS .....	157
3.3.3. Объединение результатов нескольких запросов .....	160
Контрольные вопросы .....	164
<b>4. Язык определения данных .....</b>	<b>164</b>
4.1. Домены .....	165
4.2. Создание, изменение и удаление базовых таблиц БД .....	169
4.2.1. Создание таблицы .....	169
4.2.2. Определение ограничений столбца .....	171
4.2.3. Определение ограничений на таблицу .....	174
4.2.4. Удаление таблицы БД .....	177
4.2.5. Изменение определения таблицы .....	177
4.3. Индексы .....	181
4.4. Временные таблицы .....	184
4.5. Представления .....	187
4.6. Комментарии к объектам базы данных .....	194
Контрольные вопросы .....	195

<b>5. Язык манипулирования данными</b> .....	195
5.1. Добавление новых данных .....	196
5.1.1. Однострочный запрос INSERT .....	196
5.1.2. Многострочный запрос INSERT .....	197
5.2. Обновление существующих данных .....	199
5.2.1. Простой запрос UPDATE .....	199
5.2.2. Запрос UPDATE с подзапросом .....	200
5.3. Обобщенное обновление и добавление данных .....	204
5.4. Слияние данных .....	206
5.5. Удаление существующих данных .....	208
5.5.1. Простой запрос DELETE .....	208
5.5.2. Запрос DELETE с подзапросом .....	209
5.6. Обновление представлений .....	212
Контрольные вопросы .....	221
<b>6. Процедурный язык</b> .....	221
6.1. Основы разработки модулей на PSQL .....	222
6.1.1. Переменные .....	222
6.1.2. Условные операторы .....	224
6.1.2.1. Оператор ветвления IF .....	224
6.1.2.2. Оператор WHILE .....	225
6.1.3. Курсоры в PSQL .....	225
6.1.3.1. Неявный курсор .....	226
6.1.3.2. Явный курсор .....	228
6.1.4. SQL сценарии .....	229
6.1.5. Генераторы .....	232
6.1.6. Исключительные ситуации .....	234
6.2. Хранимые процедуры .....	236
6.2.1. Определение хранимых процедур .....	236
6.2.2. Процедуры выбора .....	240
6.2.3. Выполняемые процедуры .....	245
6.3. Триггеры .....	252
6.3.1. Триггеры DML .....	252
6.3.1.1. Определение триггера .....	252
6.3.1.2. Примеры поддержания ссылочной целостности .....	258
6.3.1.3. Модификация и удаление триггера .....	261
6.3.2. Триггеры базы данных .....	262
6.4. Выполняемые блоки .....	264
Контрольные вопросы .....	265
<b>7. Защита данных</b> .....	265
7.1. Управление доступом к данным .....	266
7.1.1. Требования к безопасности данных .....	266
7.1.2. Привилегии доступа и передача привилегий .....	267
7.1.3. SQL роли .....	271
7.1.4. Отмена привилегий .....	273
7.1.5. Привилегии на представления .....	275
7.2. Транзакции .....	277
7.2.1. Понятие транзакции .....	277
7.2.2. Восстановление данных .....	278
7.2.3. Восстановление системы .....	282
7.2.4. Параллелизм .....	285
Контрольные вопросы .....	288
<b>Приложение А</b> .....	289

Приложение Б .....	296
Список литературы .....	310

## Предисловие

Предлагаемое учебное пособие написано на основе многолетнего личного опыта автора, полученного в результате разработки, внедрения, эксплуатации и сопровождения информационных систем для подразделений ОАО «Газпром», жилищно-коммунальных и других организаций, а также чтения лекций, проведения лабораторных практикумов, руководства курсового и дипломного проектирования в Рязанском государственном радиотехническом университете.

Учебное пособие адресовано в первую очередь начинающим пользователям технологии реляционных баз данных в архитектуре «клиент-сервер» (как программистам, так и администраторам баз данных, а также конечным пользователям реляционных СУБД) и содержит всю необходимую информацию для эффективного применения SQL на практике.

Все примеры выполнены на SQL СУБД Firebird в среде IBExpert. Несмотря на то, что примеры выполнены для реализации SQL в СУБД Firebird 2.1, они также, за небольшим исключением, будут работать в любой из промышленных реляционных СУБД в архитектуре «клиент-сервер».

Заинтересованные читатели на <http://www.abonentplus.ru/sqltest/> могут, как проверить полученные теоретические знания SQL, так и приобрести практические навыки, выполнив предлагаемый лабораторный практикум по всем разделам учебного пособия.

В создании настоящего учебного пособия в той или иной степени участвовало много заинтересованных лиц. Хочется от души поблагодарить заведующего кафедрой автоматизированных систем управления д-ра техн. наук, профессора Нечаева Г.И. за всестороннее содействие, аспиранта Шубенкова Е.Е. – за материалы и советы по содержанию пособия, студентов Булыгину В.Л. – за оказание практической помощи в оформлении и Шувырденкова Б.А. – за участие в разработке образовательного Internet-ресурса. Автор также глубоко признателен рецензентам, чьи ценные замечания позволили улучшить качество учебного пособия.

## Введение

Изложение материала настоящего учебного пособия строится в соответствии с рекомендуемым порядком изучения языка структурированных запросов (SQL). В пособии описываются самые последние версии запросов SQL, соответствующие стандарту SQL2003, приводится реализация этих запросов для платформы Firebird 2.1.

В первой главе даются общие минимальные сведения из систем баз данных, необходимые для понимания всего последующего материала. Здесь излагаются основные понятия и принципы организации реляционных баз данных на примере учебной базы данных, описывается процесс проектирования базы

данных, рассматриваются особенности работы с базами данных в архитектуре «клиент-сервер».

Вторая глава является кратким введением в язык SQL. В этой главе приводится определение основных объектов базы данных, описываются функции и достоинства структурированного языка запросов, приводится классификация запросов, анализируются различные формы языка. Также в данной главе рассматриваются особенности функционирования и состав СУБД Firebird.

В третьей главе рассматриваются возможности SQL по выборке данных из базы данных. Здесь изучаются практически все указанные в стандарте SQL средства выборки данных и, кроме того, представлены некоторые дополнительные возможности, реализованные в СУБД Firebird.

В четвертой главе описывается язык определения данных SQL, который позволяет создавать, изменять и удалять основные объекты базы данных.

В пятой главе рассматривается, каким образом осуществляется манипулирование данными в уже созданной базе данных.

Шестая глава знакомит читателей с особенностями использования процедурного SQL. На процедурном SQL создаются хранимые процедуры, триггеры и выполняемые блоки, позволяющие реализовать многие задачи в рамках СУБД, не обращаясь к прикладному программированию в другой среде.

В седьмой главе рассматриваются вопросы защиты данных. Излагаются общие правила разграничения доступа пользователей к объектам базы данных, рассматриваются методы управления доступом и описывается использование механизма транзакций.

Материал сопровождается большим количеством как простых, так и достаточно сложных примеров на учебной базе данных, являющейся сокращенным вариантом базы данных информационной системы «Абонент+», коллектив разработчиков которой многие годы возглавляет автор. Система «Абонент+» достаточно эффективно используется в нескольких регионах Российской Федерации для информационного обеспечения деятельности газораспределительных организаций, региональных компаний по реализации газа и других жилищно-коммунальных предприятий.

В конце каждой главы приводятся контрольные вопросы, с помощью которых можно проверить усвоение теоретического материала, изложенного в соответствующей части учебного пособия.

В приложении А приведено описание учебной базы данных, а в приложении Б – скрипт для ее создания.

## **1. Реляционная модель данных**

Неотъемлемой частью современной повседневной жизни стали базы данных, для поддержки которых требуется некоторый организационный метод, или механизм. Такой механизм называется системой управления базами данных (СУБД).

**База данных (БД)** – совместно используемый набор логически связанных

данных (и их описание), предназначенный для удовлетворения информационных потребностей пользователей.

**Система управления базами данных (СУБД)** – программное обеспечение (ПО), с помощью которого пользователи могут определять, создавать и поддерживать базу данных, а также получать к ней контролируемый доступ.

Ключевую роль при обеспечении эффективного хранения данных играют методы поддержания логических связей между данными. По способам организации связей выделяют различные модели данных. Рассматриваемая в настоящем учебном пособии СУБД Firebird, как и подавляющее большинство современных СУБД, относится к реляционным системам.

В этой главе излагаются основные понятия и принципы организации реляционных БД, а также описываются базовые подходы к проектированию реляционных БД. Используется понятие реляционной модели данных, рассматриваются структурные составляющие модели, достаточно кратко описываются процессы нормализации отношений и проектирования реляционных БД на логическом и физическом уровнях. Кроме того, в данной главе приводятся особенности организации работы с БД в архитектуре «клиент-сервер».

## **1.1. Определение реляционной модели**

Реляционная модель данных впервые была предложена американским математиком Коддом в 1970 году [1]. Фундаментальным понятием реляционной БД является отношение. Это отражено и в общем названии подхода – термин реляционный (relational) происходит от relation (отношение). На физическом уровне отношения представляют собой таблицы. В реляционной модели все данные представлены в виде простых таблиц, разбитых на строки и столбцы. К сожалению, практическое определение понятия «реляционная база данных» оказалось гораздо более расплывчатым, чем точное математическое определение, данное этому термину Коддом в 1970 году. Поставщики СУБД реализовывали в своих продуктах лишь некоторые черты реляционных систем, и, фактически, потенциальные возможности и смысл реляционного подхода искажались.

В ответ на это в 1985 году Кодд написал статью, где сформулировал 12 правил, которым должна удовлетворять любая база данных, претендующая на звание реляционной. Приведенные ниже двенадцать правил Кодда считаются определением реляционной СУБД [1, 2].

1. *Правило информации.* Вся информация в базе данных должна быть представлена исключительно на логическом уровне и только одним способом - в виде значений, содержащихся в таблицах.
2. *Правило гарантированного доступа.* Логический доступ ко всем и к каждому элементу данных (атомарному значению) в реляционной базе данных должен обеспечиваться путём использования комбинации имени таблицы, первичного ключа и имени столбца.
3. *Правило поддержки недействительных значений.* В настоящей



реляционной базе данных должна быть реализована поддержка недействительных значений, которые отличаются от строки символов нулевой длины, строки пробельных символов и от нуля или любого другого числа и используются для представления отсутствующих данных независимо от типа этих данных.

4. *Правило динамического каталога, основанного на реляционной модели.* Описание базы данных на логическом уровне должно быть представлено в том же виде, что и основные данные, чтобы пользователи, обладающие соответствующими правами, могли работать с ним с помощью того же реляционного языка, который они применяют для работы с основными данными.
5. *Правило исчерпывающего подъязыка данных.* Реляционная система может поддерживать различные языки и режимы взаимодействия с пользователем (например, режим вопросов и ответов). Однако должен существовать, по крайней мере, один язык, операторы которого можно представить в виде строк символов, в соответствии с некоторым четко определенным синтаксисом и который в полной мере поддерживает следующие элементы:
  - определение данных;
  - определение представлений;
  - обработку данных (интерактивную и программную);
  - условия целостности;
  - идентификация прав доступа;
  - границы транзакций (начало, завершение и отмена).
6. *Правило обновления представлений.* Все представления, которые теоретически можно обновить, должны быть доступны для обновления.
7. *Правило добавления, обновления и удаления.* Возможность работать с отношением (таблицей) как с одним операндом должна существовать не только при чтении данных, но и при добавлении, обновлении и удалении данных.
8. *Правило независимости физических данных.* Прикладные программы и утилиты для работы с данными должны на логическом уровне оставаться нетронутыми при любых изменениях способов хранения данных или методов доступа к ним.
9. *Правило независимости логических данных.* Прикладные программы и утилиты для работы с данными должны на логическом уровне оставаться нетронутыми при внесении в базовые таблицы любых изменений, которые теоретически позволяют сохранить нетронутыми содержащиеся в этих таблицах данные.
10. *Правило независимости условий целостности.* Должна существовать возможность определять условия целостности, специфические для конкретной реляционной базы данных, на подъязыке реляционной базы данных и хранить их в каталоге, а не в прикладной программе.
11. *Правило независимости распространения.* Реляционная СУБД не должна зависеть от потребностей конкретного пользователя.

12. *Правило единственности.* Если в реляционной системе есть низкоуровневый язык (обрабатывающий одну запись за один раз), то должна отсутствовать возможность использования его для того, чтобы обойти правила и условия целостности, выраженные на реляционном языке высокого уровня (обрабатывающем несколько записей за один раз).

Правило 1 напоминает неформальное определение реляционной базы данных, приведенное ранее.

Правило 2 указывает на роль первичных ключей при поиске информации в базе данных. Имя таблицы позволяет найти требуемую таблицу, имя столбца позволяет найти требуемый столбец, а первичный ключ позволяет найти строку, содержащую искомый элемент данных.

Правило 3 требует, чтобы отсутствующие данные можно было представить с помощью недействительных значений (NULL).

Правило 4 гласит, что реляционная база данных должна сама себя описывать. Другими словами, база данных должна содержать набор *системных таблиц*, описывающих структуру самой базы данных.

Правило 5 требует, чтобы СУБД использовала язык реляционной базы данных, например SQL, хотя явно SQL в правиле не упомянут. Такой язык должен поддерживать все основные функции СУБД — создание базы данных, чтение и ввод данных, реализацию защиты базы данных и т.д.

Правило 6 касается представлений, которые являются виртуальными таблицами, позволяющими показывать различным пользователям различные фрагменты структуры базы данных. Это одно из правил, которое сложнее всего реализовать на практике.

Правило 7 акцентирует внимание на том, что базы данных по своей природе ориентированы на множества. Оно требует, чтобы операции добавления, удаления и обновления можно было выполнять над множествами строк. Это правило предназначено для того, чтобы запретить реализации, в которых поддерживаются только операции над одной строкой.

Правила 8 и 9 означают отделение пользователя и прикладной программы от низкоуровневой реализации базы данных. Они утверждают, что конкретные способы реализации хранения или доступа, используемые в СУБД, и даже изменения структуры таблиц базы данных не должны влиять на возможность пользователя работать с данными.

Правило 10 гласит, что язык базы данных должен поддерживать ограничительные условия, налагаемые на вводимые данные и действия, которые могут быть выполнены над данными.

Правило 11 гласит, что язык базы данных должен обеспечивать возможность работы с распределенными данными, расположенными на других компьютерных системах.

И, наконец, правило 12 предотвращает использование других возможностей для работы с базой данных, помимо языка базы данных, поскольку это может нарушить ее целостность.

Однако можно сформулировать и более простое определение.

**Реляционной** называется база данных, в которой все данные, доступные пользователю, организованы в виде прямоугольных таблиц, а все операции над данными сводятся к операциям над этими таблицами.

В настоящем пособии используется учебная реляционная база данных, которая представляет собой очень упрощенный пример информационной модели системы «Абонент+», используемой для информационного обеспечения деятельности газораспределительных организаций и региональных компаний по реализации газа [3]. Полное описание таблиц учебной базы данных и содержащихся в ней данных приведено в приложении А. Далее будут рассмотрены основные понятия реляционных баз данных на примере учебной базы данных.

## 1.2. Таблицы

В реляционной базе данных информация организована в виде реляционных таблиц, разделённых на строки и столбцы, на пересечении которых содержатся значения данных [4].

**Таблица** – это некоторая регулярная структура, состоящая из конечного набора однотипных записей.

Таблица отражает тип объекта реального мира (сущность). Строки соответствуют экземпляру объекта, конкретному событию или явлению. Столбцы соответствуют атрибутам (признакам, характеристикам, параметрам) объекта, события, явления. У каждой таблицы имеется уникальное имя внутри базы данных, описывающее её содержимое.

У каждого столбца в таблице есть своё имя, которое обычно служит заголовком столбца. Все столбцы в одной таблице должны иметь уникальные имена, однако разрешается присваивать одинаковые имена столбцам, расположенным в различных таблицах. В реляционной модели данных атрибуты отношений не упорядочены, т.е. обращение к полям всегда происходит по именам, а не по расположению. Однако в языке SQL допускается индексное указание столбцов таблиц, при этом столбцы рассматриваются в порядке слева направо (их порядок определяется при создании таблицы) [1].

В любой таблице всегда есть как минимум один столбец. В стандарте ANSI/ISO не указывается максимально допустимое число столбцов в таблице, однако почти во всех коммерческих СУБД этот предел существует. В СУБД Firebird этот предел составляет 32767 столбцов.

В реляционной модели данных для обозначения строки отношения используется понятие кортеж. Представлением кортежа на физическом уровне является строка таблицы базы данных. Строки таблицы не имеют имен и определённого порядка. В таблице может содержаться любое количество строк. Вполне допустимо существование таблицы с нулевым количеством строк. Такая таблица называется пустой. Пустая таблица сохраняет структуру, определённую её столбцами, просто в ней не содержатся данные. Как правило, не накладывается ограничений на количество строк в таблице, и во многих

СУБД размер таблиц ограничен лишь свободным дисковым пространством компьютера. В других СУБД имеется максимальный предел, однако он весьма высок - около двух миллиардов строк, а иногда и больше.

Проиллюстрируем более наглядно структуру одной из таблиц учебной базы данных (см. приложение А). На рис. 1.1 приведена структура таблицы Abonent, содержащей сведения об абонентах газораспределительной организации.

	ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
Данные об абоненте Конюхов В.С.	005488	3	4	1	АКСЕНОВ С.А.	556893
	015527	3	1	65	КОНЮХОВ В.С.	761699
	080047	8	39	36	ШУБИНА Т.П.	257842
	080270	6	35	6	ТИМОШКИНА Н.Г.	321002
	080613	8	35	11	ЛУКАШИНА Р.М.	254417
Данные об абоненте Шмаков С.В.	115705	3	1	82	МИЩЕНКО Е.В.	769975
	126112	4	7	11	МАРКОВА В.П.	683301
	136159	7	39	1	СВИРИНА З.А.	350003
	136160	4	9	15	ШМАКОВ С.В.	982222
	136169	4	7	13	ДЕНИСОВА Е.К.	680305
	443069	4	51	55	СТАРДУБЦЕВ Е.В.	683014
	443690	7	5	1	ТУЛУПОВА М.И.	214833

Номер лицевого счета абонента      Идентификатор улицы, на которой проживает абонент      Номер дома      Номер квартиры      ФИО абонента      Номер телефона абонента

**Рис. 1.1.** Структура реляционной таблицы Abonent

Каждая горизонтальная строка этой таблицы представляет отдельную физическую сущность - одного абонента. Двенадцать строк таблицы вместе представляют всех абонентов газораспределительной организации. Все данные, содержащиеся в конкретной строке таблицы, представляют собой набор значений атрибутов конкретного абонента, который описывается этой строкой.

Каждый вертикальный столбец таблицы представляет совокупность значений конкретного атрибута объекта. Например, в столбце AccountCD содержатся уникальные номера лицевого счета абонентов. В столбце Phone содержатся телефонные номера абонентов.

Значение данных представляет собой действительные данные, содержащиеся в каждом элементе данных. На пересечении каждой строки с каждым столбцом таблицы содержится в точности одно значение данных. Например, в строке, представляющей абонента КОНЮХОВ В.С., в столбце Fio содержится значение 'КОНЮХОВ В.С.'. В столбце AccountCD той же строки содержится значение '015527', которое является номером лицевого счета абонента КОНЮХОВ В.С. Все значения, содержащиеся в одном и том же столбце, являются данными одного типа. Например, в столбце Fio содержатся только слова, а в столбце StreetCD содержатся целые числа, представляющие идентификаторы улиц. В реляционной модели данных общая совокупность значений, из которой берутся действительные значения для определенных атрибутов (столбцов) называется *доменом* [1]. Доменом столбца Fio, например, является множество фамилий абонентов. Каждый столбец всегда определяется на одном домене.

В реляционных базах данных домен определяется путем задания, как минимум, некоторого базового типа данных, к которому относятся элементы

домена, а часто также и произвольного логического выражения, применяемого к элементам этого типа данных (ограничения домена).

В учебной базе данных определены следующие домены:

- *Boolean (Логический): SMALLINT*. Поля, определяемые на этом домене, могут принимать только целочисленные значения, равные 0 или 1. Это достигается наложением в домене условия проверки (CHECK) на принимаемые этим доменом значения.
- *Money (Деньги): NUMERIC(15,2)*. Этот домен предназначен для определения в таблицах полей, хранящих денежные суммы.
- *PKField (Поле ПК): INTEGER*. Этот домен предназначен для определения первичных ключей таблиц. Ограничение обязательности данных (NOT NULL) на этот домен не наложено. Оно накладывается при объявлении первичного ключа таблицы. Это сделано для того, чтобы можно было определить внешний ключ на этом домене без условия NOT NULL.
- *TMonth (Месяц): SMALLINT*. Этот домен предназначен для определения в таблицах полей, содержащих номера месяцев. Целочисленные значения в таком поле могут находиться в диапазоне 1...12.
- *TYear (Год): SMALLINT*. Этот домен предназначен для определения полей, содержащих номер года. Целочисленные значения могут принимать значения в диапазоне 1990...2100.

### 1.3. Первичные ключи

Поскольку строки в реляционной таблице не упорядочены, нельзя выбрать строку по ее номеру в таблице. В таблице нет "первой", "последней" или "тринадцатой" строки. Тогда каким же образом можно указать в таблице конкретную строку, например строку для абонента с именем Аксенов С.А.?

**Ключевым элементом данных** называется такой элемент, по которому можно определить значения других элементов данных.

В реляционной базе данных в каждой таблице есть один или несколько столбцов, значения в которых во всех строках разные. Этот столбец (столбцы) называется первичным ключом таблицы.

**Первичный ключ** – это атрибут или группа атрибутов, которые единственным образом идентифицируют каждую строку в таблице.

Вернемся к рассмотрению таблицы Abonent учебной базы данных (рис. 1.1). На первый взгляд, первичным ключом таблицы Abonent могут служить и столбец AccountCD, и столбец Fio. Однако в случае если будут зарегистрированы два абонента с одинаковыми ФИО, то столбец Fio больше не сможет исполнять роль первичного ключа. На практике в качестве первичных ключей таблиц обычно следует выбирать идентификаторы, такие как уникальный номер лицевого счета абонента (AccountCD в таблице Abonent), идентификатор улицы (StreetCD в таблице Street) и т.д.

Если в таблице нет полей, значения в которых уникальны, для создания первичного ключа в нее обычно вводят дополнительное поле, значениями которого СУБД может распоряжаться по своему усмотрению.

Если первичный ключ представляет собой комбинацию столбцов, то такой первичный ключ называется *составным*.

*Вторичные ключи* устанавливаются по полям, которые часто используются при поиске или сортировке данных. В отличие от первичных ключей, поля для вторичных ключей могут содержать не уникальные значения.

## 1.4. Внешние ключи

Столбец одной таблицы, значения в котором совпадают со значениями столбца, являющегося первичным ключом другой таблицы, называется внешним ключом. Обычно в качестве первичного и внешнего ключей используются столбцы с одинаковыми именами из двух различных таблиц.

Внешний ключ, как и первичный ключ, тоже может представлять собой комбинацию столбцов. На практике внешний ключ всегда будет составным (состоящим из нескольких столбцов), если он ссылается на составной первичный ключ в другой таблице. Очевидно, что количество столбцов и их типы данных в первичном и внешнем ключах совпадают.

Если таблица связана с несколькими другими таблицами, она может иметь несколько внешних ключей. На рис. 1.2 показана таблица Request с тремя внешними ключами, ссылающимися на таблицы Abonent, Executor и Disrepair.

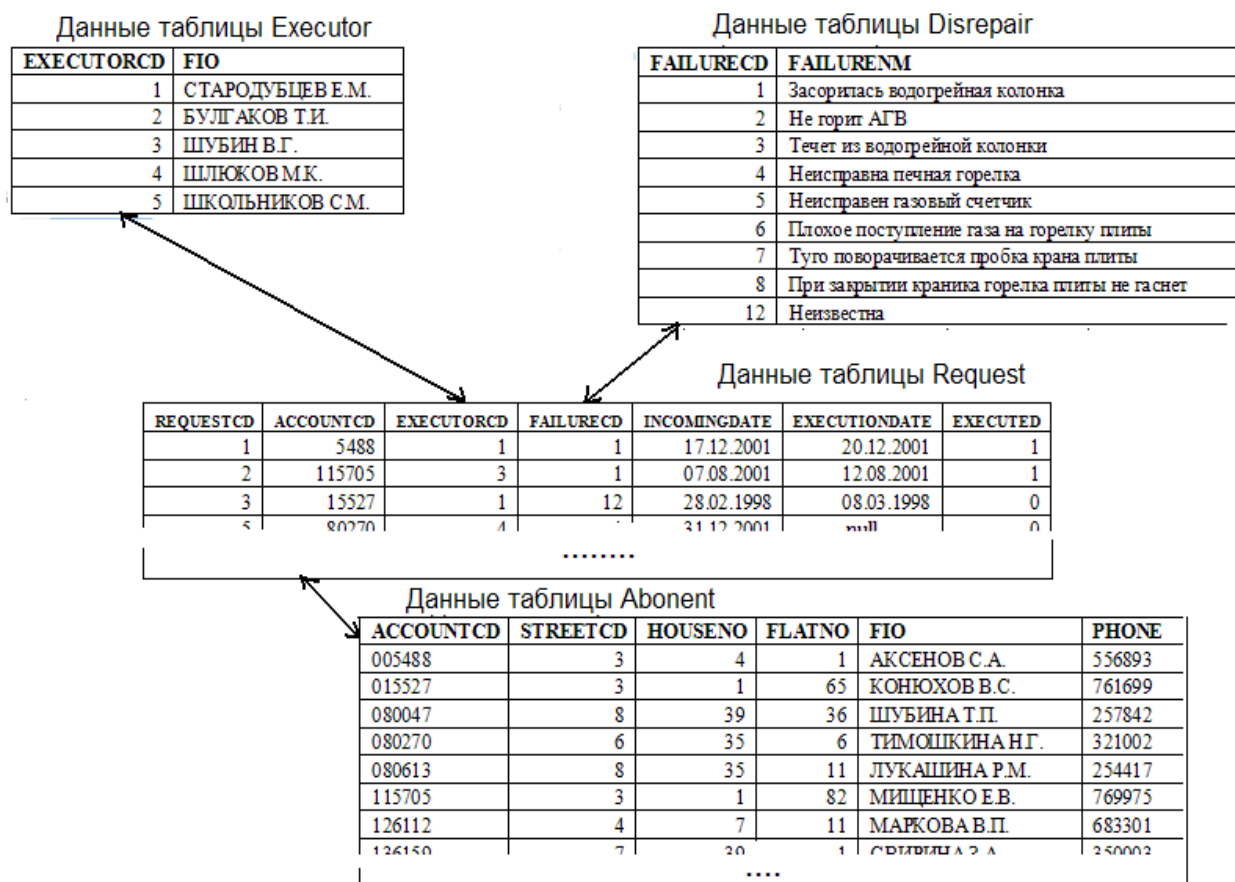


Рис. 1.2. Множественные отношения родитель-потомок в реляционной базе данных

Столбец AccountCD является внешним ключом для таблицы Request, ссылающимся на первичный ключ таблицы Abonent, и связывает каждую ремонтную заявку с абонентом, подавшим ее. Столбец ExecutorCD является внешним ключом для таблицы Request, ссылающимся на первичный ключ таблицы Executor, и связывает каждую заявку с назначенным на ее выполнение исполнителем. Столбец FailureCD является внешним ключом для таблицы Request, ссылающимся на первичный ключ таблицы Disrepair, и связывает каждую заявку с типом неисправности газового оборудования.

Внешние ключи являются неотъемлемой частью реляционной модели, поскольку реализуют отношения между таблицами базы данных.

## 1.5. Связи между таблицами

В реляционных базах данных между таблицами, как уже было отмечено, существуют связи (отношения). Если между некоторыми сущностями существует связь, то факты из одной сущности ссылаются или некоторым образом связаны с фактами из другой сущности. Связь работает путем сопоставления первичного ключа одной таблицы (родительской сущности) с элементом внешнего ключа другой таблицы (дочерней сущности) [5]. Первичный и соответствующий ему внешний ключ помогают реализовать отношение родитель-потомок между таблицами. В базе данных нужно хранить только актуальные, значимые связи.

Связи могут различаться по типу связи (идентифицирующая, не идентифицирующая, полная и неполная категория, неспецифическая связь), по мощности связи, допустимости пустых (NULL) значений.

Связь называется **идентифицирующей**, если экземпляр дочерней сущности идентифицируется (однозначно определяется) через ее связь с родительской сущностью. Атрибуты, составляющие первичный ключ родительской сущности, при этом входят в первичный ключ дочерней сущности. Дочерняя сущность при идентифицирующей связи всегда является зависимой.

Связь называется **не идентифицирующей**, если экземпляр дочерней сущности идентифицируется иначе, чем через связь с родительской сущностью. Атрибуты, составляющие первичный ключ родительской сущности, при этом входят в состав не ключевых атрибутов дочерней сущности.

*Мощность связи* представляет собой отношение количества экземпляров родительской сущности к соответствующему количеству дочерней сущности. По мощности связи выделяют отношения «один к одному», «один ко многим», «многие ко многим».

При связи «один к одному» одной строке родительской таблицы может соответствовать не более одной строки дочерней таблицы (и наоборот). Такая связь создается, если оба связанных столбца являются первичными ключами или имеют ограничение, обеспечивающее их уникальность. Связи этого типа встречаются редко, поскольку связанную подобным образом информацию обычно удается поместить в одной таблице.

«Один ко многим» - наиболее распространенный вид связи. При этом типе связи одной строке родительской таблицы может соответствовать множество строк дочерней таблицы, но любой строке дочерней таблицы может соответствовать только одна строка родительской таблицы.

Обратимся к учебной базе данных. Все связи между таблицами учебной базы данных являются не идентифицирующими с мощностью «один ко многим». Рассмотрим, например, связь «один ко многим» между таблицами Street и Abonent (рис. 1.3).

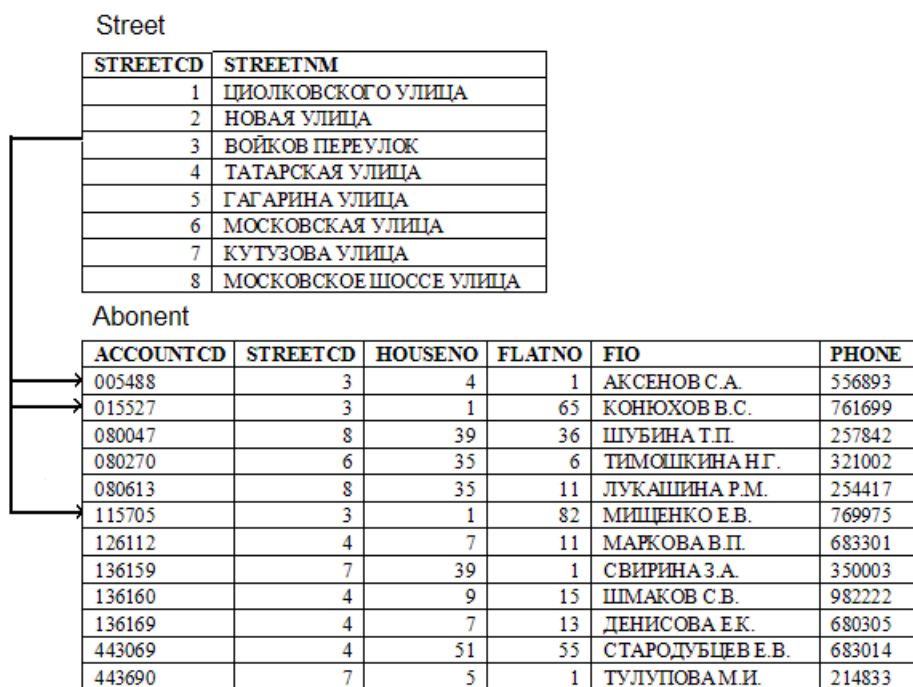


Рис. 1.3. Связь «один ко многим» между таблицами Street и Abonent

Из рис. 1.3 следует, в столбце StreetCD таблицы Abonent содержится идентификатор улицы, на которой проживает абонент. Столбец StreetCD в таблице Abonent представляет собой внешний ключ, ссылающийся на одноименный столбец таблицы Street. Доменом этого столбца (множеством значений, которые могут в нем храниться) является множество идентификаторов улиц, содержащихся в столбце StreetCD таблицы Street. Мощность отношения – «один ко многим», так как на одной и той же улице может проживать (и проживает) множество абонентов, но каждый абонент проживает только на одной определенной улице. Наименование улицы, на которой проживает, например, абонент АКСЕНОВ С.А., можно узнать, определив значение столбца StreetCD в строке таблицы Abonent со значением в столбце Fio, равным 'АКСЕНОВ С.А.' (число 3), и затем отыскав в таблице Street строку с таким же значением в столбце StreetCD (улица ВОЙКОВ ПЕРЕУЛОК). Например, чтобы найти всех абонентов, проживающих на улице ВОЙКОВ ПЕРЕУЛОК, следует запомнить значение столбца StreetCD для этой улицы (число 3), а потом просмотреть таблицу Abonent и найти все строки, в столбце StreetCD которых содержится число 3 (это строки для абонентов с



номерах лицевых счетов '005488', '015527' и '115705').

Таким образом, отношение «один ко многим», существующее между улицами и проживающими на них абонентами, в реляционной модели реализовано в виде одинаковых значений данных, хранящихся в двух таблицах. Все отношения, существующие между таблицами реляционной базы данных, реализуются в таком виде.

При связи «многие ко многим» (неспецифическое отношение) одной строке родительской таблицы может соответствовать множество строк дочерней таблицы (и наоборот). Такая связь создается с помощью третьей таблицы, первичный ключ которой состоит из внешних ключей таблиц, связанных отношением «многие ко многим».

## 1.6. Нормализация отношений

Процесс нормализации был впервые предложен Коддом в 1972 году [1, 2]. Этот процесс основан на понятии *функциональной зависимости*. По определению «функциональная зависимость – это такая связь между атрибутами В и А одного и того же отношения, когда каждому значению А соответствует только одно значение В» [2]. Атрибут А называют *детерминантом*. Детерминанты могут быть составными, т.е. представлять собой не единичные атрибуты, а группы, состоящие из двух и более атрибутов. Нормализация обычно приводит к разделению одной таблицы на две или более таблиц, соответствующих требованиям нормальных форм. Общепринятыми считаются пять нормальных форм. Сначала было предложено только три вида нормальных форм: первая (1НФ), вторая (2НФ) и третья (3НФ). Затем Бойсом и Коддом в 1974 году было сформулировано более строгое определение третьей нормальной формы, которое получило название нормальной формы Бойса-Кодда (НФБК).

Вслед за НФБК появились определения четвертой (4НФ) и пятой (5НФ) нормальных форм в 1977 и в 1979 годах. Однако на практике эти нормальные формы более высоких порядков используются редко.

Каждая последующая форма удовлетворяет требованиям предыдущей. Если следовать только первому правилу нормализации, то данные будут представлены в 1НФ. Если данные удовлетворяют третьему правилу нормализации, они будут находиться в 3НФ (а также во 1НФ и 2НФ) и т.д.

Таким образом, каждая последующая форма предъявляет больше требований к данным, чем предыдущая.

**Первая нормальная форма** требует, чтобы на любом пересечении строки и столбца находилось единственное значение, которое должно быть атомарным (неделимым). В таблице, удовлетворяющей 1НФ, не должно быть повторяющихся групп.

Обратимся к учебной базе данных. Предположим, что данные из таблиц Abonent и Street ранее содержались в одной общей ненормализованной таблице. Вид такой таблицы представлен на рис. 1.4.

Повторяющиеся группы значений

ACCOUNTCD	STREETCD	STREETNM	HOUSENO	FLATNO	FIO	PHONE
005488		→3 ВОЙКОВ ПЕРЕУЛОК	4	1	АКСЕНОВ С.А.	556893
015527		→3 ВОЙКОВ ПЕРЕУЛОК	1	65	КОНОХОВ В.С.	761699
080047		8 МОСКОВСКОЕ ШОССЕ УЛИЦА	39	36	СЕРОВА Т.П.	257842
080270		6 МОСКОВСКАЯ УЛИЦА	35	6	ТИМОШКИНА Н.Г.	321002
080613		8 МОСКОВСКОЕ ШОССЕ УЛИЦА	35	11	ЛУКАШИНА Р.М.	254417
115705		→3 ВОЙКОВ ПЕРЕУЛОК	1	82	МИЩЕНКО Е.В.	769975
126112		4 ТАТАРСКАЯ УЛИЦА	7	11	МАРКОВА В.П.	683301
136159		7 КУТУЗОВА УЛИЦА	39	1	СВИРИНА Э.А.	350003
136160		4 ТАТАРСКАЯ УЛИЦА	9	15	ШМАКОВ С.В.	982222
136169		4 ТАТАРСКАЯ УЛИЦА	7	13	ДЕНИСОВА Е.К.	680305
443069		4 ТАТАРСКАЯ УЛИЦА	51	55	СТАРДУБЦЕВ Е.В.	683014
443690		7 КУТУЗОВА УЛИЦА	5	1	ТУЛУПОВА М.И.	214833

**Рис. 1.4.** Ненормализованная таблица

Путем разбиения этой таблицы на две можно получить таблицы *Abonent* и *Street*, удовлетворяющие требованиям 1НФ. Все остальные таблицы учебной базы данных также удовлетворяют требованиям 1НФ.

**Вторая нормальная форма** основана на понятии *полной функциональной зависимости*. Атрибут *B* называется полностью функционально зависимым от атрибута *A*, если атрибут *B* функционально зависит от полного значения атрибута *A* и не зависит от какого-либо подмножества атрибута *A*.

Отношение находится во 2НФ, если оно находится в 1НФ и каждый его атрибут, не входящий в состав первичного ключа, функционально полно зависит от первичного ключа. Другими словами, второе правило нормализации требует, чтобы любой неключевой столбец зависел от всего первичного ключа, а не от его отдельных компонентов. Это правило относится к случаю, когда первичный ключ образован из нескольких столбцов. Первичные ключи всех таблиц из учебной базы данных являются простыми (состоят из одного столбца), поэтому все таблицы находятся не только в 1НФ, но и однозначно во 2НФ.

**Третья нормальная форма** основана на понятии *транзитивной зависимости*. Если для атрибутов *A*, *B* и *C* некоторого отношения существуют зависимости *C* от *B* и *B* от *A*, то говорят, что атрибут *C* транзитивно зависит от атрибута *A* через атрибут *B*.

Отношение находится в 3НФ, если оно находится в 1НФ и 2НФ, и в нем не существует транзитивных зависимостей неключевых атрибутов от первичного ключа. Другими словами, *третья нормальная форма* требует, чтобы ни один неключевой столбец не зависел бы от другого неключевого столбца. Любой неключевой столбец должен зависеть только от столбца первичного ключа.

Рассмотрим, например, зависимости между столбцами в таблице *PaySumma* учебной БД. Например, столбец *PaySum* в этой таблице не зависит от столбца *AccountCD*, так как одному абоненту соответствует множество оплаченных сумм. Также столбец *PaySum* не зависит от столбца *PayDate*, так как на одну дату может приходиться несколько оплаченных сумм и т.д. Таким образом,

между неключевыми столбцами нет функциональных зависимостей и, следовательно, нет транзитивных зависимостей этих столбцов от первичного ключа. Таблица PaySumma и все остальные таблицы учебной базы данных удовлетворяют требованиям 3НФ.

**Нормальная форма Бойса-Кодда** учитывает функциональные зависимости, в которых участвуют все потенциальные ключи отношения, а не только его первичный ключ. Для отношения с единственным потенциальным ключом 3НФ и НФБК эквивалентны.

Отношение находится в НФБК тогда и только тогда, когда каждый его детерминант является потенциальным ключом.

**Четвертая нормальная форма** связана с понятием *многозначной зависимости*. В случае многозначной зависимости, существующей между атрибутами А, В и С некоторого отношения, для каждого значения А имеется набор значений атрибута В и набор значений атрибута С. Однако входящие в эти наборы значения атрибутов В и С не зависят друг от друга.

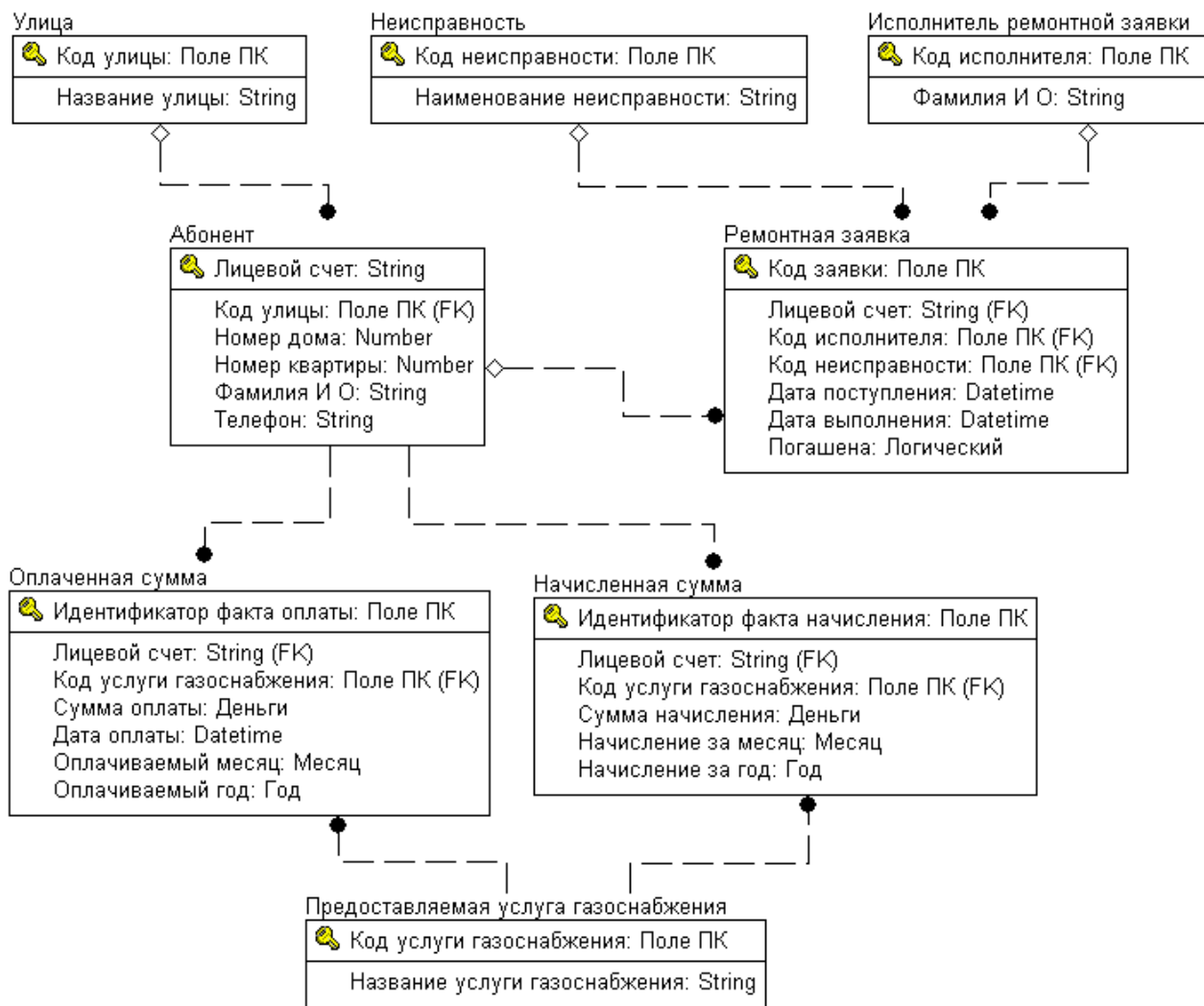
Отношение находится в 4НФ, если оно находится в НФБК и не содержит многозначных зависимостей.

**Пятой нормальной формой** называется отношение, которое не содержит *зависимостей соединения*. Зависимость соединения – это такая ситуация, при которой декомпозиция отношения может сопровождаться генерацией ложных строк при обратном соединении декомпозированных отношений с помощью операции естественного соединения.

## 1.7. Модели базы данных на логическом и физическом уровнях

Как уже было сказано, реляционная модель представляет базу данных в виде множества взаимосвязанных отношений. Перед созданием базы данных выполняется проектирование ее структуры на логическом и физическом уровнях. Такое проектирование выполняется с помощью средств автоматизированного проектирования информационных систем (CASE-средства). В настоящее время имеется множество средств автоматизации разработки, предназначенных как для разработки баз данных, так и для разработки клиентских приложений. Одной из наиболее распространенных программ для проектирования баз данных является ERwin [6]. С помощью ERwin можно выполнить проектирование на логическом и физическом уровне, а также создать базу данных на сервере.

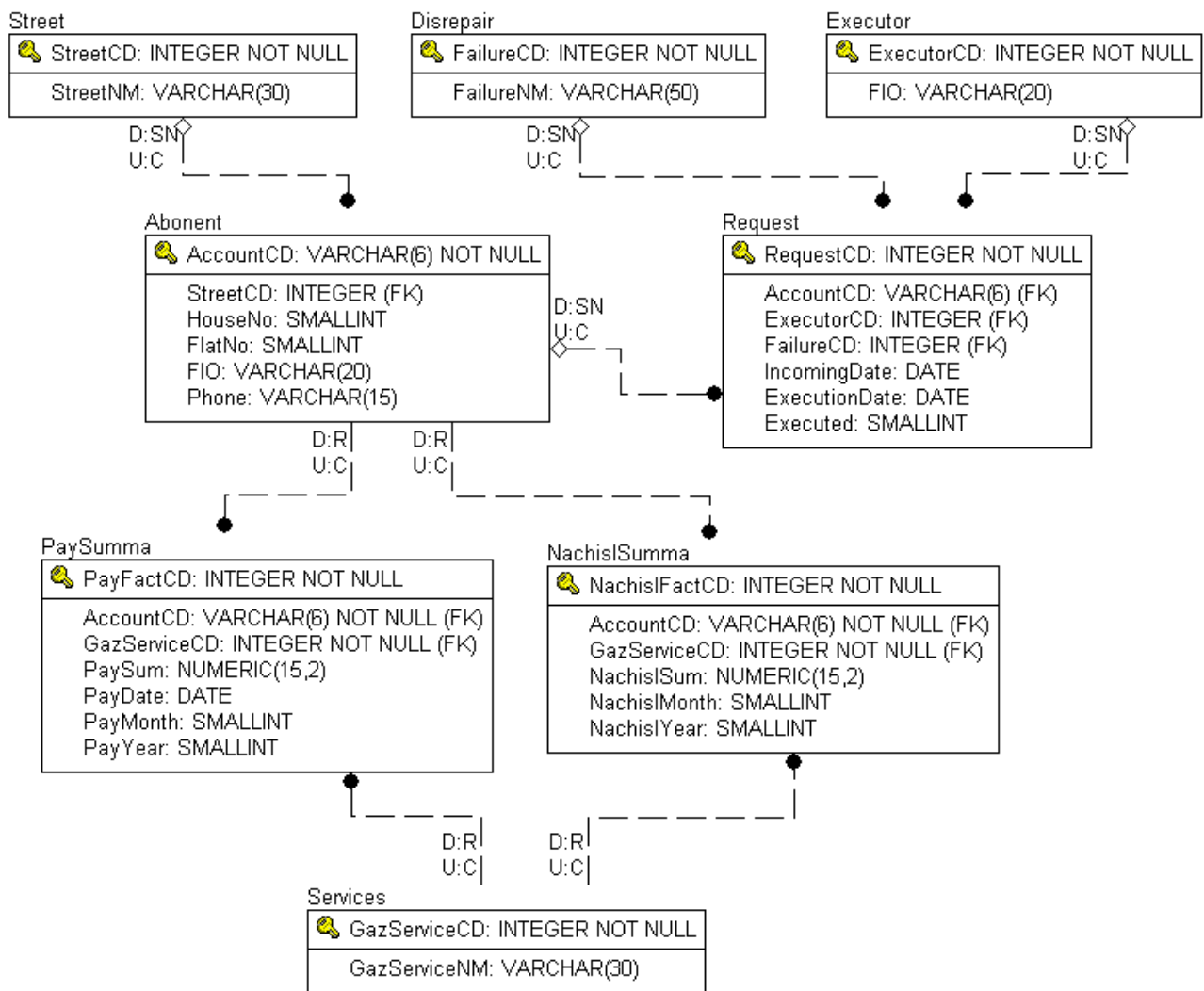
На логическом уровне проектирование выполняется путем выделения сущностей (Entity), атрибутов сущностей (Attribute) и взаимосвязей между сущностями. Модель «сущность-связь» (Entity-Relationship Model, или ER-модель) была разработана Ченом (Chen) в 1976 году с целью упрощения задачи проектирования баз данных [2]. Логическая модель независима от особенностей физической реализации объекта. На рис. 1.5 представлены отношения между всеми сущностями учебной базы данных в виде ER модели на логическом уровне.



**Рис. 1.5.** ER модель учебной БД на логическом уровне

Логическая модель данных является источником информации для физического проектирования. Физическое проектирование базы данных предусматривает принятие решения о способах реализации модели на основе конкретной СУБД. ERwin автоматически создает имена таблиц и столбцов на основе имен соответствующих сущностей и атрибутов, учитывая максимальную длину имени и другие синтаксические ограничения, накладываемые СУБД. Тип данных каждого столбца при переходе на физический уровень будет соответствовать типу данных, допустимому в конкретной СУБД. Между фазами физического и логического проектирования всегда имеется обратная связь, поскольку решения, принятые на этапе физического проектирования, могут потребовать некоторого пересмотра логической модели данных.

Модель учебной базы данных на физическом уровне представлена на рис. 1.6.



**Рис. 1.6.** ER модель учебной БД на физическом уровне

ER-диаграмма учебной БД на физическом уровне содержит восемь таблиц. Столбец, расположенный в верхней части изображения таблицы и отделенный горизонтальной чертой, является первичным ключом.

Рядом с именем каждого столбца указывается тип данных этого столбца, или, другими словами, столбец задается на определенном домене.

Предложение NOT NULL, указанное при описании столбца, накладывает ограничение, согласно которому в данном столбце недопустимо хранение NULL значений.

Указанное в определении столбца обозначение (FK) говорит о том, что данный столбец является внешним ключом, т.е. ссылается на родительскую таблицу и может содержать только те значения, которые имеются в первичном ключе родительской таблицы.

Наличие связей (Relationship) или ссылок между таблицами обозначается линиями. Конец линии с черным кружком прикрепляется к дочерней таблице, а другой конец линии прикрепляется к родительской таблице. Если у родительской таблицы линия имеет белый ромб, то это означает, что внешний ключ дочерней таблицы может содержать нулевые значения. Если линия

пунктирная, то это говорит о не идентифицирующей связи между таблицами. Если линия сплошная, то связь – идентифицирующая. Не идентифицирующая связь говорит о том, что значение первичного ключа родительской таблицы полностью не идентифицирует (определяет) значения в строке дочерней таблицы, связанной через внешний ключ со значением данного первичного ключа.

## 1.8. Целостность данных

Для пользователей информационной системы недостаточно, чтобы база данных просто отражала объекты реального мира. Важно, чтобы такое отражение было однозначным и непротиворечивым. В этом случае говорят, что база данных удовлетворяет условию *целостности*.

Для того чтобы гарантировать корректность и взаимную непротиворечивость данных, на базу данных накладываются ограничения, называемые **ограничениями целостности**. Ограничения целостности (целостная составляющая) реляционной модели можно разделить на две группы – требование целостности сущностей и требование целостности ссылок.

Первое из требований — требование **целостности сущности** — означает, что первичный ключ должен полностью идентифицировать каждую сущность, а поэтому не допускается наличие неопределенных (null) значений в составе первичного ключа. Требование целостности сущностей также подразумевает отсутствие полей с множественным характером значений атрибута, что обеспечивается нормализацией таблиц-отношений.

Требование **целостности ссылок** заключается в том, что внешний ключ не может быть указателем на несуществующую строку в таблице, т.е. для любой записи с конкретным значением внешнего ключа должна обязательно существовать связанная запись в родительской таблице с соответствующим значением первичного ключа. Ограничения целостности реализуются с помощью специальных средств, о которых речь пойдет далее.

Рассмотрим отношения ссылочной целостности и ограничения NOT NULL, которые отражены в ER модели учебной БД.

Внешние ключи AccountCD и GazServiceCD таблиц PaySumma и NachislSumma ссылаются на первичные ключи таблиц Abonent и Services соответственно. Эти внешние ключи объявлены как NOT NULL (нет белого ромбика у линии связи, примыкающей к родительской таблице), т.е. в таблицах PaySumma и NachislSumma не могут существовать записи с пустыми значениями полей AccountCD и GazServiceCD. Записи в этих таблицах могут существовать только в том случае, если в справочнике абонентов (таблица Abonent) и в справочнике услуг газоснабжения (таблица Services) есть абонент и услуга газоснабжения, которые указаны в полях внешних ключей AccountCD и GazServiceCD. Для всех связей рассматриваемых в данный момент внешних ключей определены условия ссылочной целостности: D:R и U:C. Это, например, означает, что удаление абонента из справочника абонентов будет запрещено, если в таблице PaySumma или NachislSumma существуют записи,

связанные с этим абонентом через механизм внешних ключей, а при обновлении первичного ключа таблицы Abonent будет выполнено «каскадное» обновление соответствующих внешних ключей AccountCD в таблицах PaySumma и NachislSumma.

В справочнике абонентов может отсутствовать информация об улице, на которой проживает абонент, т.е. значение внешнего ключа StreetCD таблицы Abonent может быть NULL. Аналогично для ремонтной заявки (запись в таблице Request) в общем случае может быть неизвестно, от какого абонента она принята (внешний ключ AccountCD в этом случае содержит NULL значение), какая неисправность должна быть исправлена в этой ремонтной заявке (внешний ключ FailureCD имеет значение NULL) и кто должен ее выполнить (внешний ключ ExecutorCD имеет значение NULL).

Для рассмотренных связей внешних ключей установлены следующие правила ссылочной целостности: D:SN и U:C.

D:SN означает, что при удалении соответствующей записи в родительской таблице значение внешнего ключа будет установлено NULL.

Пример для правила U:C приведен при рассмотрении внешних ключей таблиц PaySumma и NachislSumma и означает каскадное обновление внешних ключей при обновлении соответствующих им первичных ключей в родительской таблице.

Правила ссылочной целостности указываются рядом с линией связи внешнего ключа у родительской таблицы.

## **1.9. Архитектура «клиент-сервер»**

В связи с расширением рынка информационных услуг производители программного обеспечения выпускают все более интеллектуальные, а значит, и объемные программные комплексы. Многие организации и отдельные пользователи часто не могли разместить приобретенные продукты на собственных ЭВМ. При размещении БД на персональном компьютере, который не находится в сети, БД всегда используется в монопольном режиме. Однако так как БД отражает информационную модель реальной предметной области, она растет по объему и резко расширяется число задач, решаемых с ее использованием, и в соответствии с этим увеличивается количество приложений, работающих с единой БД. Компьютеры объединяются в локальные сети, и необходимость распределения приложений, работающих с единой базой данных по сети, является несомненной.

В системах обработки данных на базе локальных вычислительных сетей (ЛВС), как правило, компьютеры не являются равноправными. Каждый из них имеет свое, отличное от других, назначение, играет свою роль. Некоторые компьютеры в сети владеют и распоряжаются информационно-вычислительными ресурсами, такими как процессоры, файловая система, почтовая служба, служба печати, база данных. Другие же компьютеры имеют возможность обращаться к этим службам, пользуясь услугами первых. Для таких систем характерной чертой является то, что процессы обработки

информации частично выполняются в месте ее получения. Наиболее ресурсоемкие процессы обработки информации происходят на центральном мощном компьютере. Компьютер, управляющий тем или иным ресурсом, принято называть **сервером** этого ресурса, а компьютер, желающий им воспользоваться, – **клиентом**.

**Сервер** – логический процесс, который обеспечивает обслуживание запросов других процессов. Сервер не посылает результатов запрашивающему процессу до тех пор, пока не придет запрос на обслуживание. После инициирования запроса управление синхронизацией обслуживания и связей становится функцией самого сервера.

Конкретный сервер определяется видом ресурса, которым он владеет. В качестве ресурса сервера применительно к технологии БД выступает сама БД.

**Сервер баз данных** - фактически обычная СУБД, принимающая запросы по локальной сети и возвращающая результаты, т.е. основное назначение - обслуживать запросы клиентов, связанные с обработкой данных. Высокопроизводительный интеллектуальный сервер баз данных является сердцевинной любой СУБД.

Другими словами, *сервер базы данных* – это логический процесс, отвечающий на обработку запросов к базе данных. Его техническое качество в решающей степени определяет главные характеристики системы, такие как производительность, надежность, безопасность и т.д.

**Рабочая станция** предназначена для непосредственной работы пользователя или категории пользователей и обладает ресурсами, соответствующими локальным потребностям данного пользователя.

**Клиент** – это процесс, посылающий серверу запрос на обслуживание. Главной особенностью является то, что клиент может начать транзакцию связи с сервером, а сервер никогда не начинает транзакцию связи с клиентом.

Функцией клиента являются инициирование установления связи, запрос конкретного вида обслуживания, получение от сервера результатов и подтверждение окончания обслуживания. Хотя клиент может запросить синхронное или асинхронное уведомление об окончании обслуживания, он сам не управляет синхронизацией и связью.

В построении различных систем возможен ряд модификаций в зависимости от того, какие функции прикладной программы (или, проще, приложения) будут реализованы в программе-клиенте, а какие - в сервере.

Применительно к технологиям баз данных функции стандартного интерактивного приложения разделяются на четыре группы, имеющие различную природу [7]:

- функции ввода и отображения данных;
- прикладные функции, определяющие основные алгоритмы решения задач приложения и характерные для данной предметной области;
- фундаментальные функции хранения и управления информационными ресурсами;
- служебные функции, играющие роль связок между функциями первых трех групп.



Эта условная классификация показывает, как могут быть распределены отдельные задачи между серверным и клиентским процессом.

В соответствии с этим в любом приложении выделяются следующие логические компоненты [8]:

- **компонент представления**, реализующий функции первой группы;
- **прикладной компонент**, поддерживающий функции второй группы;
- **компонент доступа к информационным ресурсам**, поддерживающий функции третьей группы, а также вводятся и уточняются соглашения о способах их взаимодействия (протокол взаимодействия).

Различия в реализациях моделей обработки данных определяются четырьмя следующими факторами:

- в какие виды программного обеспечения интегрированы каждый из этих компонентов;
- какие механизмы программного обеспечения используются для реализации функций всех трех групп;
- как логические компоненты распределяются между компьютерами в сети;
- какие механизмы используются для связи компонентов между собой.

Различают два основных типа архитектуры СУБД: архитектура файлового сервера и архитектура «клиент-сервер».

*Модель файлового сервера (File Server - FS)* является базовой для локальных сетей персональных компьютеров. Недавно она была, да и сейчас остается, популярной среди отечественных разработчиков, использовавших такие СУБД, как Foxpro, Clipper, Clarion, Paradox и т.д.

Один из компьютеров в сети считается файловым сервером и предоставляет услуги по обработке файлов другим компьютерам. На клиентах (нескольких персональных компьютерах) выполняется как прикладная программа, так и копия СУБД. Когда прикладная программа обращается к базе данных, СУБД направляет запрос на файловый сервер на операции дискового ввода-вывода. В этом запросе указаны файлы, где находятся запрашиваемые данные. В ответ на запрос файловый сервер направляет по сети требуемые блоки данных. СУБД, получив его, выполняет над данными действия, которые были декларированы в прикладной программе.

Таким образом, организация и управление БД целиком ложатся на клиентов, а сама БД представляет собой набор файлов в одном или нескольких каталогах на сетевом сервере.

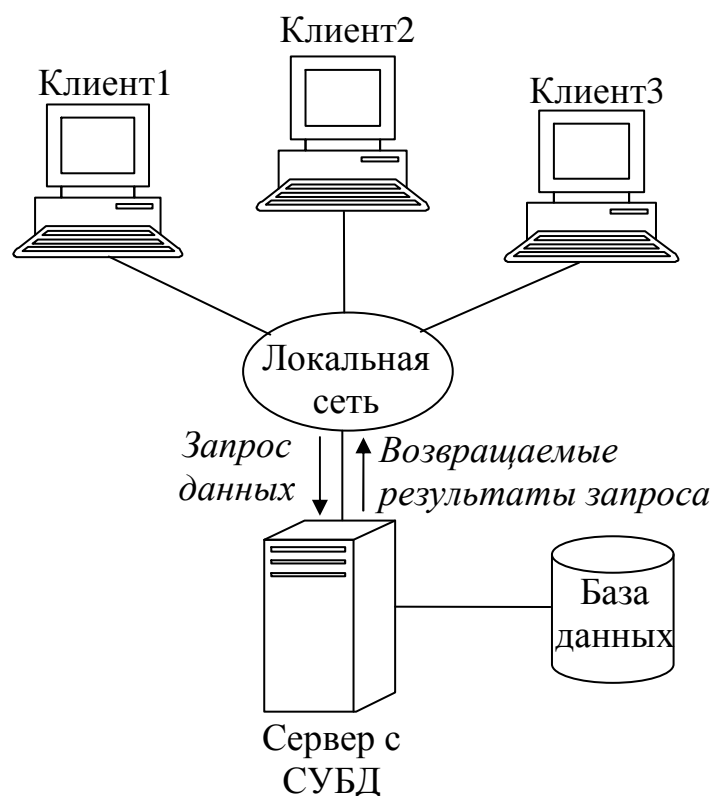
К технологическим недостаткам модели относят следующие [7]:

- высокая загрузка сети (высокий сетевой трафик) из-за передачи множества файлов, необходимых приложению, и, как следствие, увеличение требований к аппаратным мощностям пользовательского компьютера;
- узкий спектр операций манипуляции с данными («данные - это файлы»);
- отсутствие надежных средств обеспечения безопасности доступа к данным (защита самой операционной системы только на уровне файловой системы);

- управление параллельностью, восстановлением и целостностью усложняется, так как доступ к одним и тем же файлам могут осуществлять сразу несколько экземпляров СУБД.

Архитектура «клиент-сервер» была разработана с целью устранения недостатков, имеющих в модели файлового сервера. «Клиент-сервер» – это модель взаимодействия компьютеров в сети. Существуют два варианта архитектуры «клиент-сервер»: традиционная **двухуровневая** и **трехуровневая**, более пригодная для работы в среде Web.

На рис. 1.7 представлена общая схема построения систем с двухуровневой архитектурой «клиент-сервер».



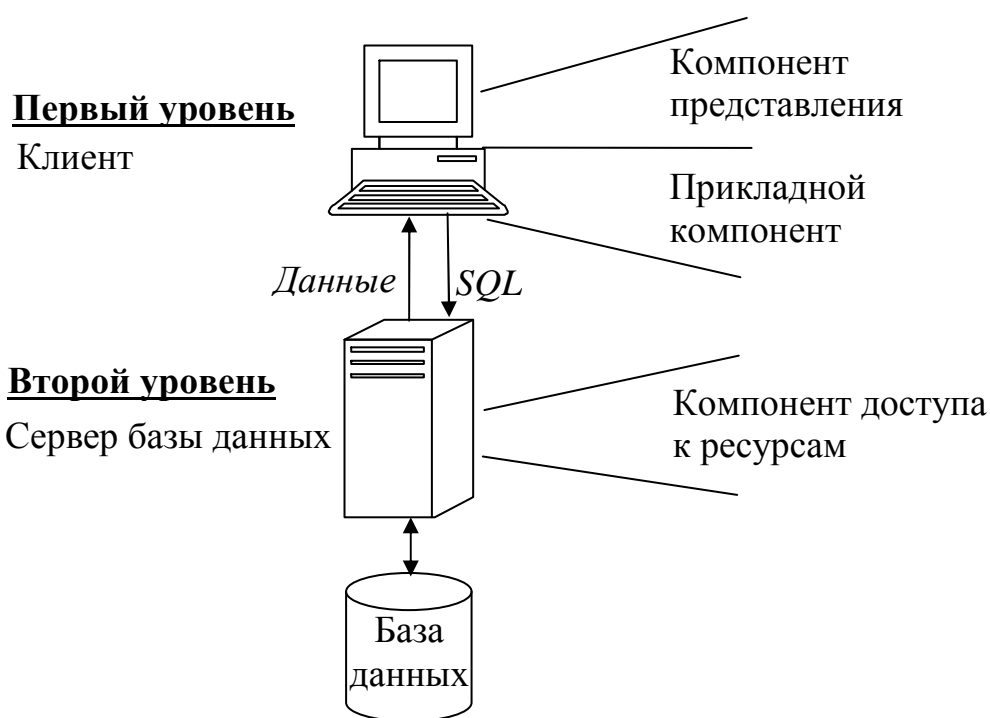
**Рис. 1.7.** Общая схема построения систем с двухуровневой архитектурой «клиент-сервер»

Исторически первая клиент-серверная система получила название **модель доступа к удаленным данным (Remote Data Access – RDA)**[7]. RDA-модель часто называют клиент-серверной архитектурой с "толстым" клиентом, поскольку в ее клиентском приложении объединены компонент представления и прикладной компонент. Модель доступа к удаленным данным представлена на рис. 1.8.

В **RDA-модели** имеется сервер баз данных. Программы компонента представления и прикладного компонента совмещены и выполняются на компьютере-клиенте. Клиент поддерживает как функции ввода и отображения данных, так и чисто прикладные функции. Доступ к информационным ресурсам обеспечивается либо операторами специального языка (например,

языка SQL для баз данных), либо вызовами функций специальной библиотеки (если имеется соответствующий интерфейс прикладного программирования - API).

Клиент направляет запросы к информационным ресурсам по сети удаленному компьютеру. На нем функционирует ядро СУБД. Оно обрабатывает запросы, выполняет предписанные в них действия и возвращает клиенту результат, оформленный как блок данных (рис. 1.8). При этом инициатором манипуляций с данными выступают программы, выполняющиеся на компьютерах-клиентах. Ядру СУБД отводится пассивная роль - обслуживание запросов и обработка данных.



**Рис. 1.8.** Модель доступа к удаленным данным

RDA-модель избавляет от недостатков, присущих как системам с централизованной архитектурой, так и системам с файловым сервером [7].

*Во-первых*, перенос компонента представления и прикладного компонента на компьютеры-клиенты существенно разгружает сервер БД, сводя к минимуму общее число процессов операционной системы. Сервер БД освобождается от несвойственных ему функций.

*Во-вторых*, процессор или процессоры сервера целиком загружаются операциями обработки данных, запросов и транзакций. Это становится возможным благодаря отказу от терминалов и оснащению рабочих мест компьютерами, которые обладают собственными локальными вычислительными ресурсами, полностью используемыми программами переднего плана.

*В-третьих*, резко уменьшается загрузка сети, так как по ней передаются от клиента к серверу не запросы на ввод-вывод (как в системах с файловым сервером), а запросы на языке SQL, их объем существенно меньше.

*Основное достоинство RDA-модели* – унификация интерфейса в виде языка SQL. Запросы, направляемые программой ядру, должны быть понятны обоим. Но в СУБД уже существует язык SQL, о котором шла речь. Поэтому целесообразно использовать его не только в качестве средства доступа к данным, но и в качестве стандарта общения клиента и сервера.

Такое общение можно сравнить с беседой нескольких человек, когда один отвечает на вопросы остальных (вопросы задаются одновременно). Причем делает это он так быстро, что время ожидания ответа приближается к нулю. Высокая скорость общения достигается, прежде всего, благодаря четкой формулировке вопроса, когда спрашивающему и отвечающему людям не нужно дополнительных консультаций по сути вопроса. Беседующие обмениваются несколькими короткими однозначными фразами, им ничего не нужно уточнять.

Но и RDA-модель не лишена ряда следующих недостатков:

- взаимодействие клиента и сервера посредством SQL-запросов существенно загружает сеть;
- удовлетворительное администрирование приложений в RDA-модели практически невозможно из-за совмещения в одной программе различных по своей природе функций (функции представления и прикладные).

Перечисленных недостатков во многом лишены СУБД, построенные по архитектуре «клиент-сервер» с «тонким» клиентом. Она получила название **модель сервера базы данных** (DataBase Server - **DBS**). В этой архитектуре клиентское приложение реализует только функцию отображения информации.

Особенности данной модели следующие:

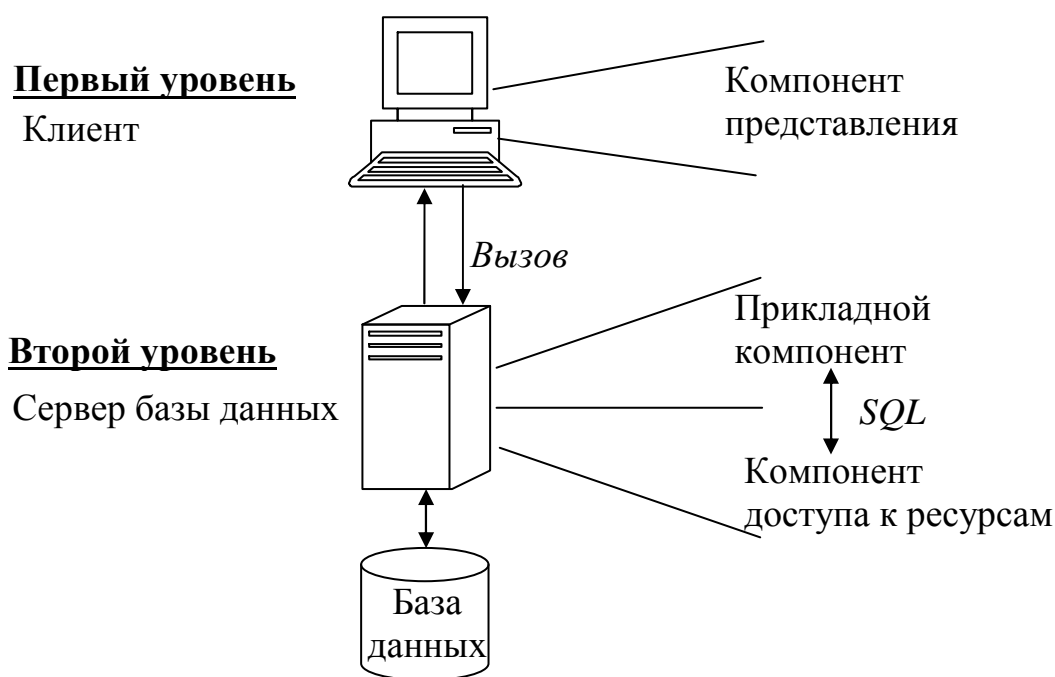
- перенос вычислительной нагрузки на сервер БД (SQL-сервер) и максимальная разгрузка клиента от вычислительной работы;
- существенное укрепление безопасности данных – как от злонамеренных, так и ошибочных изменений.

В сети один и тот же компьютер может выполнять как роль, так и сервера. Например, в информационной системе, включающей персональные компьютеры, большую ЭВМ и мини-компьютер под управлением UNIX, последний может выступать как в качестве сервера базы данных, обслуживая запросы от клиентов - персональных компьютеров, так и в качестве клиента, направляя запросы большой ЭВМ.

Этот же принцип распространяется и на взаимодействие программ. Если одна из них выполняет некоторые функции, предоставляя другим соответствующий набор услуг, то такая программа выступает в качестве сервера. Программы, которые пользуются этими услугами, принято называть клиентами. Так, ядро реляционной SQL-ориентированной СУБД часто называют сервером базы данных, или SQL-сервером, а программу, обращающуюся к нему за услугами по обработке данных, – SQL-клиентом. На рис. 1.9 представлена модель сервера базы данных.

Как и в архитектуре «файл-сервер», БД в этом случае помещается на сетевом сервере, однако программа клиента лишена возможности прямого доступа к БД. Доступ к БД регулируется специальной программой – сервером БД.

Взаимодействие сервера БД и клиента реализуется с помощью SQL-запросов, которые формирует и отправляет серверу клиент. Сервер, приняв запрос, выполняет его и возвращает результат клиенту. В клиентском приложении в основном осуществляются интерпретация полученных от сервера данных, реализация пользовательского интерфейса, а также реализация части бизнес-правил.



**Рис. 1.9.** Модель сервера базы данных

DBS-модель реализована в некоторых реляционных СУБД (Firebird, Informix, Ingres, Sybase, Oracle). Ее основу составляет механизм хранимых процедур – средство программирования SQL-сервера.

В DBS-модели компонент представления выполняется на компьютере-клиенте, в то время как прикладной компонент оформлен как набор хранимых процедур и функционирует на компьютере-сервере БД. Там же выполняется компонент доступа к данным, то есть ядро СУБД.

*Хранимые процедуры* представляют собой группу команд SQL, объединенных в один модуль. Такая группа команд, объединяющая запросы и процедурную логику (операторы присваивания, логического ветвления и т.д.), компилируется и выполняется как единое целое.

Хранимые процедуры позволяют содержать вместе с базой данных достаточно сложные программы, выполняющие большой объем работы без передачи данных по сети и взаимодействия с клиентом. Клиентское приложение обращается к серверу с командой запуска хранимой процедуры, а сервер выполняет эту процедуру и регистрирует все изменения в БД. Как правило, программы, записываемые в хранимых процедурах, связаны с обработкой данных. Тем самым база данных может представлять собой функционально самостоятельный уровень приложения, который может

взаимодействовать с другими уровнями для получения запросов или обновления данных.

*Словарь данных (системный каталог СУБД)* – это централизованное хранилище сведений об объектах, элементах данных, входящих в состав объектов, взаимосвязях между объектами, их источниках, значениях, использовании и форматах представления.

Процедуры хранятся в словаре базы данных, разделяются между несколькими клиентами и выполняются на том же компьютере, где функционирует SQL-сервер. Хранимые процедуры могут быть использованы несколькими клиентами, что существенно уменьшает дублирование алгоритмов обработки данных в разных клиентских приложениях. Язык, на котором разрабатываются хранимые процедуры, представляет собой процедурное расширение языка запросов SQL и уникален для каждой конкретной СУБД. Более подробно хранимые процедуры будут рассмотрены позднее.

Достоинства DBS-модели:

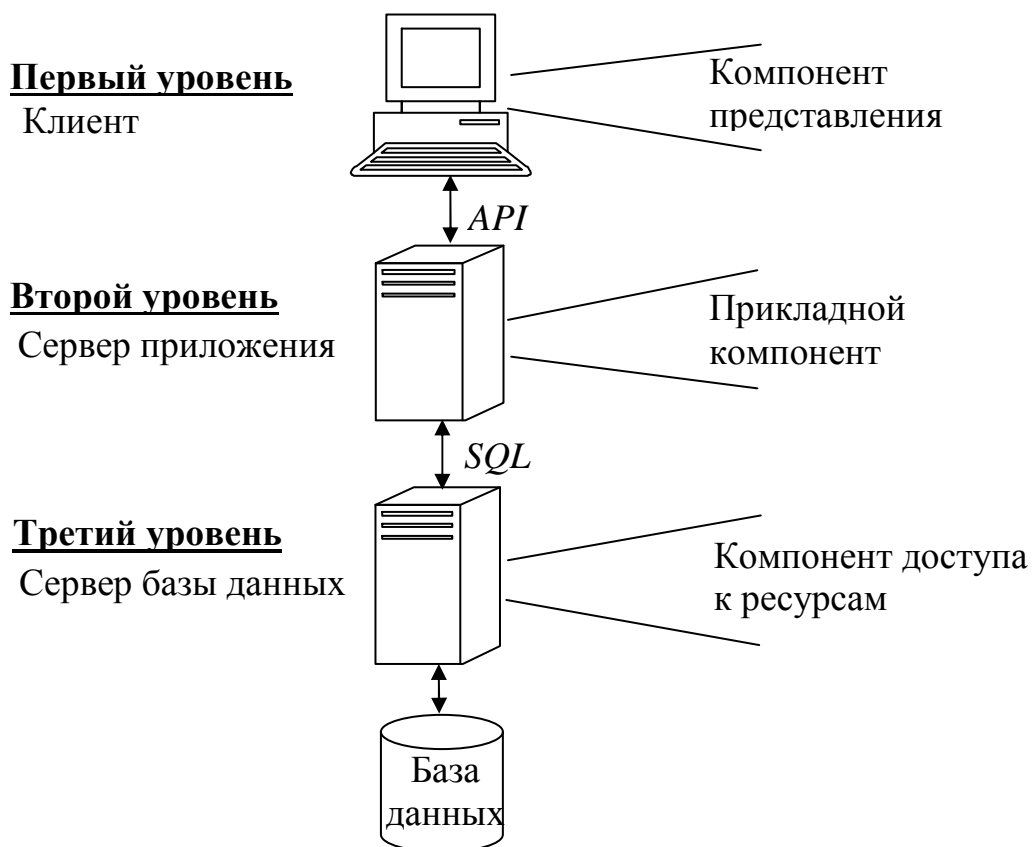
- возможность централизованного администрирования прикладных функций;
- снижение трафика (вместо SQL-запросов по сети направляются вызовы хранимых процедур);
- возможность разделения процедуры между несколькими приложениями;
- экономия ресурсов компьютера за счет использования единой созданного плана выполнения процедуры.

К недостаткам модели можно отнести ограниченность средств, используемых для написания хранимых процедур, которые представляют собой разнообразные процедурные расширения SQL, не выдерживающие сравнения по изобразительным средствам и функциональным возможностям с языками третьего поколения, такими как C или Pascal. Сфера их использования ограничена конкретной СУБД, в большинстве СУБД отсутствуют возможности отладки и тестирования разработанных хранимых процедур.

На практике часто используются смешанные модели, когда поддержка целостности базы данных и некоторые простейшие прикладные функции поддерживаются хранимыми процедурами (DBS-модель), а более сложные функции реализуются непосредственно в прикладной программе, которая выполняется на компьютере-клиенте (RDA-модель). Так или иначе, современные многопользовательские СУБД опираются на RDA- и DBS-модели и при создании ИС, предполагающем использование только СУБД, выбирают одну из этих двух моделей либо их разумное сочетание.

Необходимость масштабируемости систем по мере развития предприятий стала непреодолимым барьером для традиционной двухуровневой архитектуры клиент-сервер. Стремительно усложнявшиеся приложения требовали разворачивания их программного обеспечения на сотнях и тысячах компьютеров конечных пользователей. В результате появился трехуровневый вариант архитектуры клиент-сервер. Такая модель обработки данных получила название *модель сервера приложений (Application Server - AS)*.

Трехуровневая архитектура «клиент-сервер» представлена на рис. 1.10.



**Рис. 1.10.** Трехуровневая архитектура «клиент-сервер»

В **AS-модели** процесс, выполняющийся на компьютере-клиенте, отвечает, как обычно, за интерфейс с пользователем (то есть осуществляет функции первой группы). Обращаясь за выполнением услуг к прикладному компоненту, этот процесс играет роль клиента приложения (Application Client - AC). Прикладной компонент реализован как группа процессов, выполняющих прикладные функции, и называется сервером приложения (Application Server - AS). Все операции над информационными ресурсами выполняются соответствующим компонентом, по отношению к которому AS играет роль клиента. Из прикладных компонентов доступны ресурсы различных типов - базы данных, очереди, почтовые службы и др.

Трехуровневая архитектура довольно естественно отображается на среду Web, где Web-браузер исполняет роль «тонкого» клиента, а Web-сервер – сервера приложений.

Таким образом, можно подвести некоторые итоги по рассмотренным моделям архитектуры «клиент-сервер». RDA- и DBS-модели опираются на двухзвенную схему разделения функций. В RDA-модели прикладные функции приданы программе-клиенту. В DBS-модели ответственность за их выполнение берет на себя ядро СУБД. В первом случае прикладной компонент сливается с компонентом представления, во втором - интегрируется в компонент доступа к информационным ресурсам. В AS-модели реализована трехзвенная схема

разделения функций. Здесь прикладной компонент выделен как важнейший изолированный элемент приложения. Для его определения используются универсальные механизмы многозадачной операционной системы и стандартизованы интерфейсы с двумя другими компонентами. AS-модель является фундаментом для мониторов обработки транзакций (Transaction Processing Monitors - ТРМ), или, проще, мониторов транзакций, которые выделяются как особый вид программного обеспечения.

В заключение отметим, что часто, говоря о сервере базы данных, подразумевают как компьютер, так и программное обеспечение - ядро СУБД. При описании архитектуры «клиент-сервер» под сервером базы данных подразумевался компьютер. Далее сервер базы данных будет пониматься как программное обеспечение - ядро СУБД.

## **Контрольные вопросы**

1. Кем были предложены правила, которые считаются определением реляционной СУБД? В чем смысл этих правил?
2. Как организована информация в реляционной БД?
3. Какие виды ключей могут быть определены для таблиц БД?
4. Как реализуется отношение родитель-потомок в реляционной БД?
5. Какие существуют виды связей между таблицами?
6. На каком понятии основан процесс нормализации?
7. В каком случае таблица находится в 1НФ, 2НФ, 3НФ, НФБК, 4НФ и 5НФ?
8. На каких уровнях осуществляется проектирование БД и в чем отличие между проектированием на этих уровнях?
9. Какие средства используются в БД для поддержания целостности?
10. Что такое сервер БД и клиент? Какие функции они выполняют?
11. На какие группы можно разделить функции стандартного интерактивного приложения применительно к технологиям БД?
12. Какие существуют модели архитектуры СУБД и в чем заключаются их основные особенности?

## **2. Введение в SQL**

Настоящая глава представляет собой краткое введение в язык SQL. Изложенный в этой главе материал относится, в основном, к реализациям SQL во всех реляционных СУБД, однако некоторые особенности характерны только для СУБД Firebird, используемой в данном пособии.

Здесь приводится определение основных объектов базы данных: таблиц, представлений, хранимых процедур, триггеров и т.д. Дается определение структурированного языка запросов SQL, описываются его функции и достоинства, которые сделали SQL неотъемлемой частью современных СУБД.



Приводится классификация запросов SQL, анализируются различные формы языка и вводятся основные понятия (идентификаторы, константы, выражения и т.д.).

Рассматриваются особенности функционирования и состав СУБД Firebird, поддерживаемые типы данных, а также описываются различные утилиты для работы с БД Firebird.

Следует учесть, что последующие главы построены на основе материала данной главы. Излагаемые здесь сведения являются базовыми для правильного построения запросов SQL и понимания процесса их выполнения.

## 2.1. Объекты структуры базы данных

Рассмотрим логическую структуру реляционной базы данных.

Логическая структура определяет структуру таблиц, взаимоотношения между ними, список пользователей, хранимые процедуры, правила, умолчания и другие объекты базы данных [9]. Информацию об объектах базы данных можно получить из системных таблиц.

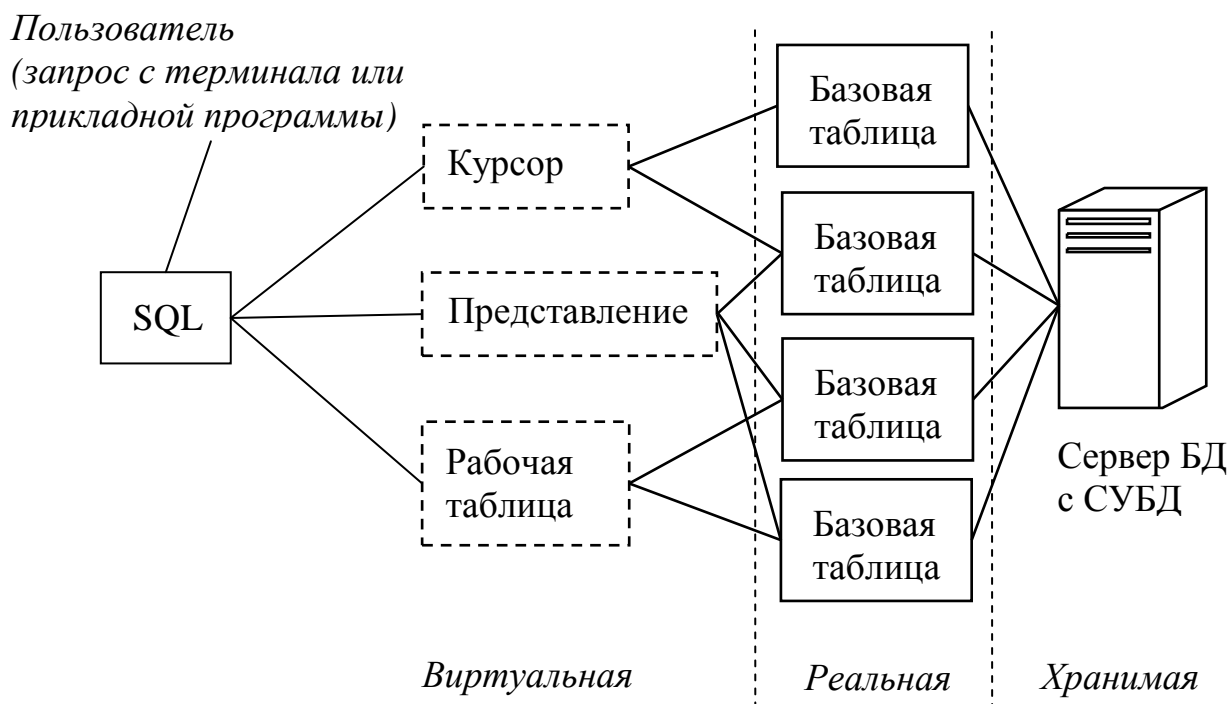
К основным объектам базы данных СУБД Firebird относятся объекты, представленные в табл. 2.1.

**Таблица 2.1.** Основные объекты базы данных

<b>Объект</b>	<b>Описание</b>
Tables	Таблицы базы данных, в которых хранятся собственно данные
Views	Просмотры (виртуальные таблицы) для отображения данных из таблиц
Stored Procedures	Хранимые процедуры
Triggers	Триггеры – специальные хранимые процедуры, вызываемые при изменении данных в таблице
User Defined function	Создаваемые пользователем функции
Indexes	Индексы – дополнительные структуры, призванные повысить производительность работы с данными
Domains	Определяемые пользователем наборы значений на основе существующих типов данных
Keys	Ключи – один из видов ограничений целостности данных
Constraints	Ограничение целостности – объекты для обеспечения логической целостности данных
Users	Пользователи, обладающие доступом к базе данных
Roles	Роли, позволяющие объединять пользователей в группы
Exceptions	Исключения – сообщения об ошибках, создаваемые пользователем
Generators	Генераторы последовательностей – специальные объекты БД для получения целочисленных значений, следующих с определенным шагом

Приведем краткий обзор основных объектов базы данных и представление БД с точки зрения пользователя.

Реляционная БД основана на табличном построении информации. База данных в восприятии пользователя представлена на рис. 2.1 [10].



**Рис. 2.1.** База данных в восприятии пользователя

Все данные в БД содержатся в объектах, называемых **таблицами**. Таблицы представляют собой совокупность каких-либо сведений об объектах, явлениях, процессах реального мира. Никакие другие объекты не хранят данные, но они могут обращаться к данным в таблице. Таблицы в SQL имеют такую же структуру, что и таблицы всех других СУБД, и содержат:

- строки; каждая строка (или запись) представляет собой совокупность атрибутов (свойств) конкретного экземпляра объекта;
- столбцы; каждый столбец (поле) представляет собой атрибут или совокупность атрибутов. Поле строки является минимальным элементом таблицы. Каждый столбец в таблице имеет определенное имя, тип данных и размер.

**Базовой** (реальной, целевой) таблицей называется таблица, для каждой строки которой в действительности имеется некоторый двойник, хранящийся в физической памяти машины.

Однако СУБД создает и использует ряд виртуальных таблиц, в которых формируются результаты запросов на получение данных из базовых таблиц. Это таблицы, которые не существуют в базе данных, но как бы существуют с точки зрения пользователя. Они называются **производными** (виртуальными) таблицами (рис. 2.1).

Базовые таблицы реально существуют, а производные - предоставляют различные способы просмотра базовых таблиц.

Таким образом, **производная** таблица – это такая таблица, которая определяется в терминах других таблиц и, в конечном счете, в терминах базовых таблиц.

К производным таблицам относятся: представления, курсоры и неименованные рабочие таблицы.

**Представление (просмотр)** – это пустая именованная таблица, определяемая перечнем тех столбцов из одной или нескольких таблиц и признаками тех их строк, которые хотелось бы в ней увидеть.

Для конечных пользователей представление выглядит как таблица, но в действительности оно не содержит данных, а лишь представляет данные, расположенные в одной или нескольких таблицах. Представления являются объектами базы данных, информация в которых формируется динамически при обращении к ним. Содержимое *представлений* выбирается из других таблиц с помощью выполнения запроса, причем при изменении значений в таблицах данные в *представлении* автоматически меняются.

Представление является как бы «окном» в одну или несколько базовых таблиц.

**Курсор** – это пустая именованная таблица, определяемая перечнем тех столбцов базовых таблиц и признаками тех их строк, которые хотелось бы в ней увидеть, и связанный с этой таблицей указатель текущей записи.

В чем различие между представлением и курсором? Для пользователя представления почти не отличаются от базовых таблиц – есть лишь некоторые ограничения при выполнении различных операций манипулирования данными. Они могут использоваться как в интерактивном режиме, так и в прикладных программах. Курсоры же созданы для процедурной работы с таблицами в хранимых процедурах, триггерах и в прикладных программах. Другими словами, курсор – это такой вид указателя, который может быть использован для перемещения по набору строк, указывая поочередно на каждую из них и обеспечивая таким образом возможность адресации к этим строкам – к одной за один раз.

В **рабочих таблицах** формируются результаты запросов на получение данных из базовых таблиц и, возможно, представлений.

Кроме объектов БД, воспринимаемых пользователем, существует ряд других важных объектов, о которых обычный пользователь может не знать.

**Хранимые процедуры** представляют собой группу команд SQL, объединенных в один модуль. Такая группа команд компилируется и выполняется как единое целое.

**Триггерами** называется специальный класс хранимых процедур, автоматически запускаемых при добавлении, изменении или удалении данных из таблицы.

**Функции** в языках программирования – это конструкции, содержащие часто исполняемый код. Функции выполняют какие-либо действия над данными и

возвращают некоторые значения. Функции могут быть встроенными или определяемыми пользователем (UDF).

**Индекс** – структура, связанная с таблицей или представлением и предназначенная для ускорения поиска информации в них. Индекс определяется для одного или нескольких столбцов, называемых индексированными столбцами. Он содержит отсортированные значения индексированного столбца или столбцов со ссылками на соответствующую строку исходной таблицы или представления. Повышение производительности достигается за счет сортировки данных. Использование индексов может существенно повысить производительность поиска, однако для хранения индексов необходимо дополнительное пространство в базе данных.

**Домены** сродни концепции «типы данных, определенные пользователем». Хотя и невозможно создать новый тип данных, в домене можно определить набор атрибутов на основе одного из существующих типов данных, присвоить ему идентификатор и после этого использовать его как параметр типа данных для определения столбца любой таблицы.

**Ограничения целостности** – механизм, обеспечивающий автоматический контроль соответствия данных установленным условиям (или ограничениям). Ограничения целостности имеют приоритет над триггерами, правилами и значениями по умолчанию. К ограничениям целостности, в частности, относятся: ограничение на значение NULL, проверочные ограничения, ограничение уникальности (уникальный ключ), ограничение первичного ключа и ограничение внешнего ключа. Последние три ограничения тесно связаны с понятием ключей.

**Исключения** – это сообщения об ошибках, создаваемые пользователем. Существуют как самостоятельный объект БД с именем, заданным пользователем при создании исключения. Используются в хранимых процедурах и триггерах, обычно после проверки какого-либо условия на истинность. Когда появляется ошибка, вызывается исключение с заданным текстом сообщения.

**Генератор последовательности** – это специальный объект БД для получения целочисленных значений, следующих с определенным шагом. В СУБД Firebird каждый генератор имеет уникальное имя и текущее значение. Данный объект БД – очень важный с точки зрения практического использования механизм порождения уникальных значений, которые могут использоваться для уникальной идентификации строк таблиц.

## 2.2. Функции SQL

*SQL* – это аббревиатура языка структурированных запросов (*Structured Query Language*). SQL является промежуточным звеном между БД и пользователем (или прикладной программой). SQL не является ни языком программирования, ни системой управления базами данных, ни отдельным программным продуктом. *SQL входит в современные СУБД.*

Нельзя, например, пойти в магазин и купить SQL, как Delphi, Microsoft Office, Microsoft Visual Studio и т.д.

SQL является инструментом, предназначенным для обработки и чтения данных, содержащихся в реляционной БД, и основан на реляционной модели данных.

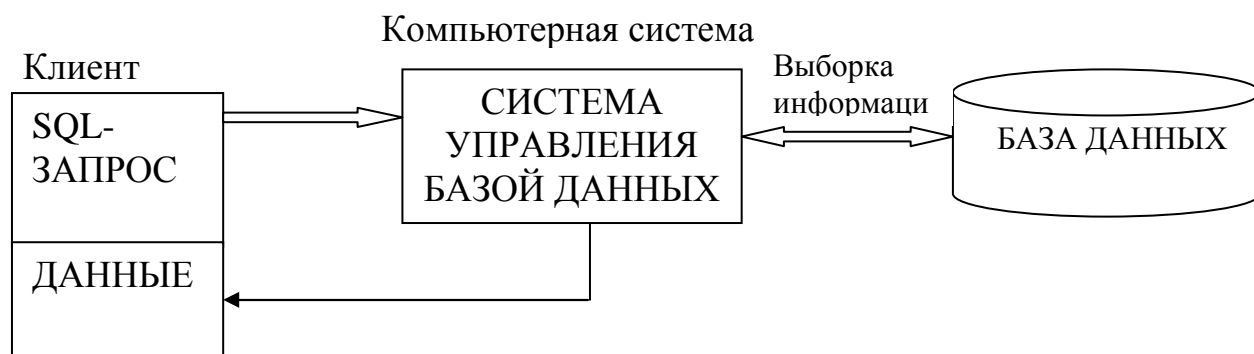
SQL является декларативным языком (на нем записывается что необходимо сделать, а не как необходимо), однако стандартный набор запросов дополняется процедурным языком, который предполагает расширение SQL средствами программирования.

SQL был разработан в 1974 году фирмой IBM.

За несколько последних лет SQL стал *единственным* языком баз данных. На сегодняшний день SQL поддерживают свыше ста СУБД, работающих как на персональных компьютерах, так и на больших ЭВМ.

На рис. 2.2 изображена упрощенная схема работы SQL при выборке информации из базы данных.

Согласно этой схеме в вычислительной системе имеется *база данных*, в которой хранятся данные.



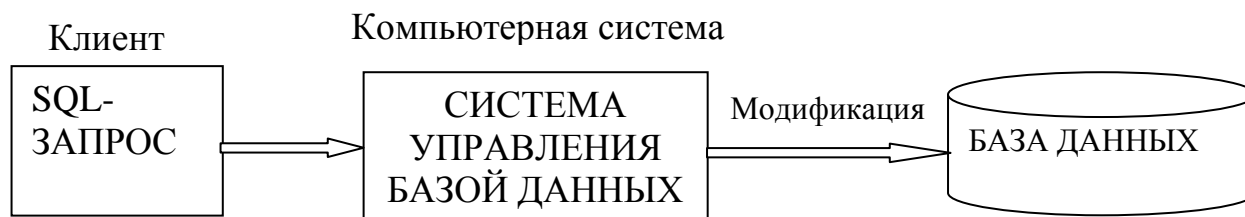
**Рис. 2.2.** Упрощенная схема работы SQL при выборке информации из базы данных

Если пользователю (клиенту) необходимо прочитать данные из базы данных, он запрашивает их у СУБД с помощью команд языка SQL. СУБД обрабатывает команду, находит требуемые данные и посылает их пользователю. Эта информация обычно выдается на экран. Ее можно также послать на принтер, сохранить в файле или представить как входные данные для другой команды, процесса или программы.

Процесс запрашивания информации и размещения результата в память называется *запросом* к базе данных (отсюда и название — структурированный язык *запросов*). *Важно*, клиент посылает SQL-запрос серверу баз данных, который возвращает лишь результат запроса, а не все данные из БД, как это происходит, например, если доступ к БД производится непосредственно из клиентского приложения.

Пользователь может не только считывать информацию из базы данных, но также модифицировать ее (добавлять, изменять или удалять уже имеющуюся).

На рис. 2.3 изображена упрощенная схема работы SQL при модифицировании информации в базе данных.



**Рис. 2.3.** Упрощенная схема работы SQL при модификации информации в базе данных

Таким образом, SQL — это неотъемлемая часть СУБД, инструмент, с помощью которого осуществляется связь клиентского приложения с реляционной базой данных.

SQL используется для реализации всех функциональных возможностей, которые СУБД предоставляет пользователю, а именно:

- *организация данных*, т.е. позволяет определять и изменять структуры представления данных и устанавливать отношения между элементами базы данных;
- *чтение данных*, т. е. предоставляет пользователю или приложению возможность читать из базы данных содержащуюся в ней информацию и пользоваться ею;
- *обработка данных*, т.е. дает возможность изменять базу данных (добавлять новые данные, удалять или обновлять уже имеющиеся);
- *управление доступом*, т.е. позволяет задавать необходимые возможности пользователя по чтению и изменению данных, а также защищать их от несанкционированного доступа;
- *совместное использование данных*, т.е. координирует совместное использование данных пользователями, работающими параллельно, чтобы они не мешали друг другу;
- *целостность данных*, т.е. обеспечивает целостность базы данных, защищая ее от разрушения из-за несогласованных изменений или отказа системы.

Какие же функции выполняет SQL?

На рис. 2.4 изображена структурная схема типичной СУБД, компоненты которой соединяются в единое целое с помощью SQL (своего рода "клея") [11].

*Ядро базы данных* является сердцевинной СУБД и выполняет следующие функции:

- отвечает за физическое структурирование и запись данных на диск;
- отвечает за физическое чтение данных с диска;
- принимает SQL-запросы от других компонентов СУБД (таких как генератор форм, генератор отчетов или модуль формирования интерактивных запросов), от пользовательских приложений и от других вычислительных систем.

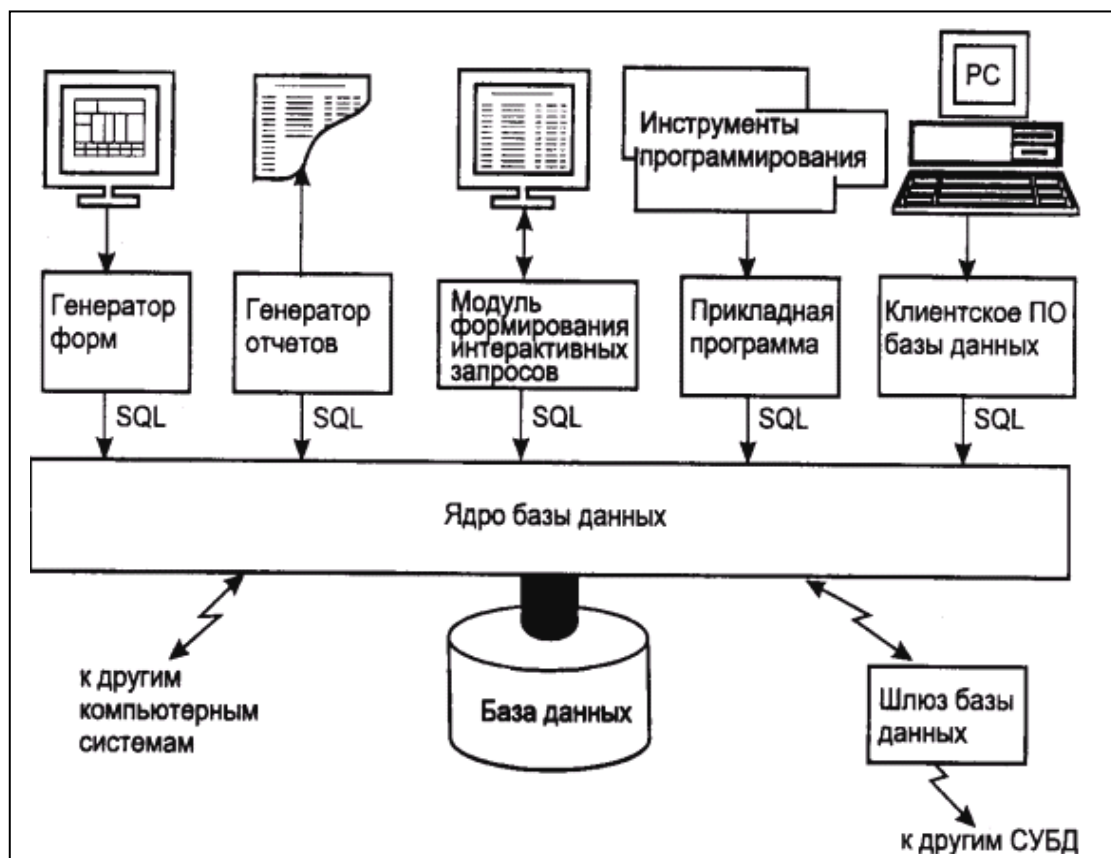


Рис. 2.4. Компоненты типичной СУБД

SQL выполняет много различных функций:

1) SQL является *языком интерактивных запросов*.

SQL обеспечивает пользователям немедленный доступ к данным. С помощью SQL пользователь может в интерактивном режиме оперативно получить ответы на самые сложные запросы, тогда как программисту потребовались бы достаточно много времени, чтобы написать для пользователя соответствующую программу. Из-за того, что SQL допускает немедленные запросы, данные становятся более доступными и могут помочь в принятии решений, делая их более обоснованными.

Пользователи вводят команды SQL также в интерактивные программы, предназначенные для чтения данных и отображения их на экране (WiSQL, IVExpert и т.д.);

2) SQL – это *язык программирования баз данных* (обеспечение программного доступа к базам данных).

Реализация в SQL концепции операций, ориентированных на табличное представление данных, позволила создать компактный язык с небольшим набором предложений. Язык SQL может использоваться как для выполнения *запросов* к данным, так и для построения прикладных программ. Для получения доступа к базе данных в программы вставляются команды SQL. Эта методика используется как в программах, написанных пользователями, так и в служебных программах баз данных (таких, как генераторы отчетов и инструменты ввода данных). Одни и те же операторы SQL используются как

для интерактивного, так и для программного доступа. Поэтому части программ, содержащие обращения к базе данных, можно вначале тестировать в интерактивном режиме, а затем встраивать в программу.

В традиционных базах данных для программного доступа используются одни программные средства, а для выполнения немедленных запросов – другие, без какой-либо связи между этими двумя режимами доступа;

3) SQL – это *язык администрирования баз данных*.

Администратор базы данных использует SQL для определения структуры базы данных;

4) SQL – это *язык создания приложений «клиент-сервер»*.

SQL используется для организации связи через локальную сеть с сервером базы данных, в котором хранятся совместно используемые данные;

5) SQL – это *язык распределенных баз данных*.

В системах управления распределенными базами данных SQL помогает распределять данные среди нескольких взаимодействующих вычислительных систем. ПО каждой системы с помощью SQL связывается с другими системами, посылая им запросы на доступ к данным;

б) SQL – это *язык шлюзов базы данных*.

В вычислительных сетях с различными СУБД SQL часто используется в *шлюзовой программе*, которая позволяет СУБД одного типа связываться с СУБД другого типа.

Таким образом, SQL является полезным и мощным инструментом, обеспечивающим людям, программам и вычислительным системам доступ к информации, содержащейся в реляционных базах данных.

### 2.3. Достоинства SQL

SQL является современным универсальным программным средством управления данными.

Успех языку SQL принесли следующие его преимущества [11].

1. *Независимость от конкретных СУБД:*

а) ведущие поставщики СУБД используют SQL, и ни одна новая СУБД, не поддерживающая SQL, не может рассчитывать на такой успех, которого достигли MS SQL, Oracle, MySQL, SyBase и Firebird;

б) реляционную базу данных и программы, которые с ней работают, в большинстве случаев можно перенести с одной СУБД на другую с минимальными доработками и переподготовкой персонала. Причем чем больше SQL конкретной СУБД соответствует стандарту, тем проще сделать переход на другую СУБД;

в) программные средства, входящие в состав СУБД для персональных компьютеров, такие, как программы для создания запросов, генераторы отчетов и генераторы приложений, работают с реляционными базами данных многих типов.

2. *Переносимость с одной вычислительной системы на другую:*

а) SQL используется в СУБД, предназначенных для различных



вычислительных систем: от персональных компьютеров и рабочих станций до локальных сетей, мини-компьютеров и больших ЭВМ;

б) однопользовательские приложения на основе SQL могут быть перенесены в более крупные системы;

в) информация из корпоративных реляционных баз данных может быть загружена в базы данных отдельных подразделений или в личные базы данных;

г) приложения для реляционных баз данных можно вначале смоделировать на экономичных персональных компьютерах, а затем перенести на дорогие многопользовательские системы.

### 3. Наличие стандартов.

Официальный стандарт языка SQL был опубликован Американским институтом национальных стандартов (American National Standards Institute — ANSI) и Международной организацией по стандартам (International Standards Organization — ISO) в 1986 году.

Затем в 1992 году он был расширен до стандарта SQL:92(SQL2).

В 1999 году появился стандарт SQL:99 (известный также как SQL3). Он характеризуется как «объектно-ориентированный SQL» и является основой нескольких объектно-реляционных систем управления базами данных (включая среди прочих ORACLE8 компании Oracle, Universal Server компании Informix, DB2 Universal Database компании IBM и Cloudscape компании Cloudscape).

В конце 2003 году был принят и опубликован новый вариант международного стандарта SQL:2003.

Эти стандарты служат как бы официальной печатью, одобряющей SQL, и они ускорили завоевание им рынка.

В настоящее время ведущими СУБД, построенными на основе SQL, являются DB2, SQL/DS, Rdb/VMS, Oracle, Ingres, Sybase, Informix, MS SQL, SQL Base, Firebird и др.

Язык SQL, соответствующий последним стандартам SQL:2003, SQL:99 (и даже SQL:92), представляет собой очень богатый и сложный язык, все возможности которого трудно сразу осознать и тем более понять.

Материал настоящего учебного пособия, посвященный языку SQL, опирается, главным образом, на наиболее поздний стандарт SQL:2003.

### 4. Реляционная основа:

а) SQL является языком реляционных баз данных, поэтому он стал популярным тогда, когда популярной стала реляционная модель представления данных;

б) табличная структура реляционной базы данных интуитивно понятна пользователям, поэтому язык SQL является достаточно простым и легким для изучения;

в) реляционная модель имеет солидный теоретический фундамент, на котором были основаны эволюция и реализация реляционных баз данных.

На волне популярности, вызванной успехом реляционной модели, SQL стал *единственным* языком для реляционных баз данных.

### 5. Высокоуровневая структура, напоминающая английский язык.

SQL-запросы выглядят как обычные английские предложения, что упрощает их изучение и понимание. Частично это обусловлено тем, что SQL-запросы *описывают* данные, которые необходимо получить, а не *определяют способ* их поиска. Таблицы и столбцы в реляционной базе данных могут иметь длинные описательные имена. В результате большинство SQL-запросов означают именно то, что точно соответствует их именам, поэтому их можно читать как простые, понятные предложения.

#### *6. Возможность различного представления данных.*

С помощью SQL создатель базы может сделать так, что различные пользователи базы данных будут видеть *различные* представления её структуры и содержимого. Например, базу данных можно спроектировать таким образом, что каждый пользователь будет видеть только данные, относящиеся к его подразделению или торговому району. Другой вариант - спроектировать базу данных так, что данные из различных её частей могут быть скомбинированы и представлены пользователю в виде одной простой таблицы.

Следовательно, представления можно использовать для усиления защиты базы данных и ее настройки под конкретные требования отдельных пользователей.

#### *7. Полноценность как языка, предназначенного для работы с базами данных.*

Первоначально SQL был задуман как язык интерактивных запросов, но сейчас он вышел далеко за рамки чтения данных.

SQL является полноценным и логичным языком, предназначенным для следующих целей:

- создание базы данных;
- управление ее защитой;
- изменение ее содержимого;
- чтение данных;
- совместное использование данных несколькими пользователями,

работающими параллельно.

Приемы, освоенные при изучении одного раздела языка, могут затем применяться в других командах, что повышает производительность работы пользователей.

#### *8. Возможность динамического определения данных.*

С помощью SQL можно динамически изменять и расширять структуру базы данных даже в то время, когда пользователи обращаются к ее содержимому. Это большое преимущество перед языками статического определения данных, которые запрещают доступ к базе данных во время изменения ее структуры.

Таким образом, SQL обеспечивает максимальную гибкость, так как дает базе данных возможность адаптироваться к изменяющимся требованиям, не прерывая работу приложения, выполняющегося в реальном масштабе времени.

#### *9. Поддержка архитектуры «клиент-сервер».*

SQL играет ключевую роль в технологии «клиент-сервер». «Клиент-сервер» – это модель взаимодействия компьютеров (сервера и клиента) в компьютерной сети. Сервер – это собственно СУБД. Он поддерживает все основные функции СУБД:

- определение данных;
- их обработку;
- защиту;
- целостность и т.д.

Клиенты – это различные приложения, выполняемые "над" СУБД. Они могут быть написанными пользователями или встроенными, т.е. предоставляемыми вместе с СУБД или сторонними поставщиками ПО.

Архитектура «клиент-сервер» позволяет существенно снизить сетевой трафик и повысить быстродействие, как персональных компьютеров, так и серверов баз данных.

SQL – естественное средство для реализации приложений «клиент-сервер». В этой роли SQL служит связующим звеном между клиентской системой, взаимодействующей с пользователем, и серверной системой, управляющей базой данных, позволяя каждой системе сосредоточиться на выполнении своих функций. Кроме того, SQL позволяет персональным компьютерам функционировать в качестве клиентов по отношению к сетевым серверам или более крупным базам данных, установленным на больших ЭВМ; это позволяет получать доступ к корпоративным данным из приложений, работающих на персональных компьютерах.

Все перечисленные факторы явились причиной того, что SQL стал стандартным инструментом для управления данными на персональных компьютерах, мини-компьютерах и больших ЭВМ. Ниже эти факторы рассмотрены более подробно.

## 2.4. SQL-сервер Firebird

Как уже было сказано, SQL используется не самостоятельно, а входит в состав различных СУБД. И хотя SQL – язык стандартизированный, в каждой конкретной СУБД он реализован по-своему. Часть команд, используемых в СУБД, точно соответствует стандарту SQL, другая часть - не присутствует в стандарте и представляет собой расширения языка для данной конкретной СУБД. Следует также учесть, что в основном СУБД не предоставляют всех возможностей, определенных стандартом.

Одной из ведущих SQL-ориентированных СУБД является Firebird. В настоящем учебном пособии используется Firebird 2.1. Созданный как проект с открытыми исходными кодами, Firebird является потомком СУБД InterBase фирмы Borland [12].

Firebird – это современная система управления реляционными базами данных (RDBMS – Relational Database Management System, СУРБД), применяемая для разработки сложных приложений на базе технологии «клиент-сервер».

SQL-сервер Firebird предназначен для хранения и обработки больших объемов данных в условиях одновременной работы с БД множества клиентских приложений. Основная цель - уменьшение риска потери или разрушения данных в случае несанкционированного доступа в многопользовательской и конкурентной среде. В настоящее время Firebird существует на таких наиболее

популярных платформах, как Windows и Linux, а также на таких платформах Unix, как FreeBSD и Mac OS X.


Firebird – это СУРБД промышленного применения, чьи возможности имеют высокий уровень соответствия стандартам SQL, при этом она реализует некоторые мощные расширения языка процедурного программирования.

Файл базы данных Firebird 2.1 на физическом уровне представляет собой один файл с расширением \*.fdb, в котором хранятся все данные, метаданные и права доступа. Файл резервной копии имеет расширение \*.fbk.

СУБД Firebird обеспечивает 2 режима работы:

- локальной машины (клиента);
- сервера БД.

Сервер Firebird работает по следующей схеме. На компьютере с ОС Windows XP/2000/NT запускается служба fbserver.exe.

Если используется ОС Windows 98, то сервер запускается как приложение. В этом случае после его загрузки на панели задач будет отображаться следующая пиктограмма: . При работе в качестве службы данная пиктограмма не отображается.

На машине (клиенте), с которой будет осуществляться доступ к серверу (т.е. к какой-либо БД под его управлением), должна быть размещена клиентская часть сервера – библиотека fbclient.dll. Следует отметить, что в более ранних версиях в качестве клиентской части сервера выступала библиотека gds32.dll, которая при инсталляции должна была быть скопирована в системный каталог Windows (предварительно в папке "system32" должны быть удалены ее копии). Эта библиотека может быть скопирована туда и сейчас (для совместимости с существующими приложениями), но фактически она просто повторяет клиентскую библиотеку fbclient.dll, расположенную в установочном каталоге Firebird 2.1 в папке bin.

Доступ к БД может осуществляться с использованием сетевого соединения (например, по протоколу TCP/IP) в случае, если производится подключение к удаленному серверу.

Локальное соединение используется, если и сервер, и приложение размещены на одном и том же компьютере. Режим сервера БД является основным.

Режим локальной машины предназначен для отладки программ и SQL-запросов (команд) при работе с БД, расположенной на том же компьютере. Для доступа к локальной БД Firebird на компьютере должна быть загружена программа Firebird Server.

Осуществление доступа к БД, расположенной на удаленном сервере, производится на компьютере с инсталлированным Firebird Client. Он посредством сети связи взаимодействует с Firebird Server, получая доступ к БД.

Для обеспечения безопасности в Firebird 2.1 используется файл БД пользователей *security2.fdb*, который содержит информацию о зарегистрированных пользователях и их паролях.

Файл конфигурации сервера *firebird.conf* содержит различные характеристики для конфигурирования сервера.

Также в состав Firebird 2.1 входит файл *aliases.conf*. В этом текстовом файле можно сопоставить конкретный путь к БД и псевдоним, чтобы затем в прикладных кодах использовать более короткий и удобный псевдоним для обращения к нужной БД.

СУБД Firebird 2.1 содержит различные утилиты командной строки для администрирования БД. Это *gbak.exe*, *nbackup.exe* (средства резервного копирования и восстановления БД для предотвращения потери данных), *isql.exe* (SQL командной строки), *gsec.exe* (для управления пользователями и их паролями), *gstat* (для сбора статистики по БД) и другие.

Кроме утилит командной строки, входящих в состав СУБД Firebird 2.1, существует достаточное количество графических инструментов администрирования БД [13].

Для разработки и администрирования БД Firebird, а также для выполнения интерактивных SQL запросов наиболее универсальным инструментом является **IBExpert** [14].

Одним из полнофункциональных продуктов для работы с Firebird является **IBAdmin** [15]. Database Designer поможет визуально представить структуру базы данных, Grant Manager поможет в управлении правами пользователей, SQL Debugger пригодится при отладке процедур и триггеров. Database Comparer будет очень полезен в том случае, когда нужно привести структуру базы клиента к самой последней версии, Dependencies Explorer поможет отследить зависимости между объектами в базе данных и т.д.

Для анализа статистики баз данных Firebird, поиска проблем в производительности базы данных, сопровождении или работы приложений используется инструмент **IBAnalyst** [12].

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.

Разработчик, пользуясь IBAnalyst, может буквально за несколько секунд увидеть все потенциальные и актуальные проблемы в своей базе данных: плохие индексы, фрагментированные таблицы, состояние транзакций и т.д., а также получить профессиональные советы о том, как решить эти проблемы.

**InterBase/Firebird Development Studio** - это мощный инструмент для разработчика баз данных под управлением серверов InterBase или Firebird [14]. Он полезен на всех стадиях разработки, начиная от начального проектирования с помощью ER-диаграмм и заканчивая обслуживанием работающей системы.

В состав пакета InterBase/Firebird Development Studio входят следующие семь приложений, каждое из которых выполняет определенную задачу.

### **1. Database Designer - дизайнер базы данных.**

ER-диаграммы очень полезны на ранних этапах разработки, и этот модуль предоставляет возможность работы с диаграммами. Но это далеко не основная его возможность. Можно вести всю разработку базы данных, не выходя из дизайнера. Проект в дизайнерах содержит все типы объектов базы данных от

таблиц до BLOB фильтров. Встроенная система контроля версий позволяет не терять ранее сделанных изменений и, конечно же, позволяет работать над одним проектом совместно. Встроенный механизм сравнения структуры баз позволяет в любой момент выполнить обновление структуры базы данных.

## **2. Database Editor - редактор базы данных.**

Редактор предназначен для самой привычной работы - редактирования объектов прямо в базе данных. Здесь же доступны отладчик хранимых процедур, инструмент получения полного скрипта базы и SQL монитор. Все редакторы кода поддерживают динамическую подсветку ошибок. Рефакторинг также может проводиться здесь. Имеется возможность переименовывать поля таблиц с автоматическим обновлением всех зависимых объектов. Можно переименовать саму таблицу - при этом будут сохранены все данные и обновлены все зависимые объекты.

## **3. Query Analyzer- анализатор запросов.**

Это отдельный инструмент для работы с запросами. Именно здесь показывается детальная статистика выполнения запроса. План запроса показывается в графическом виде, что существенно упрощает понимание того, как сервер выполняет запрос и какие индексы с какой характеристикой он использует. Приложение хранит историю запросов, что позволяет в любой момент вернуться к запросу, который пробовался, к примеру, 6 попыток назад. Можно открыть несколько редакторов одновременно и работать с несколькими базами данных одновременно. Наиболее часто используемые запросы можно сохранять в Избранном.

## **4. Database Comparer - сравнение структуры БД.**

Основное предназначение данной программы - это обновление структуры базы данных по эталону из другой базы. Удобное средство произвести апгрейд базы у клиента. Существует возможность автоматизировать процесс с применением ключей командной строки.

## **5. Administrative Console - консоль Администратора.**

Консоль администратора предназначена для выполнения резервного копирования или восстановления, проверки базы на наличие ошибок, создания пользователей и тому подобное. Следует обратить внимание на анализатор статистики базы – там информация, выдаваемая сервером, показана в более удобном виде.

## **6. Perfomance Monitor - монитор производительности.**

Наиболее полезна данная утилита для тех, кто использует InterBase 7.x и старше. Этот сервер содержит системные временные таблицы, в которых представлена масса информации о текущем состоянии сервера. Для всех остальных программа показывает только основные параметры, такие как объем памяти, занимаемый сервером, число пользователей, число использованных буферов.

## **7. Maintenance Service Manager - консоль управления планировщиком Time to Backup.**

В комплекте с Interbase/Firebird Development Studio поставляется сервис планировщика от Time to Backup. А данное приложение позволяет управлять

этим сервисом и настраивать регулярное выполнения резервного копирования с различными настройками. Поддерживается автоматическая подстановка текущей даты/времени в имя файла, номера файла по порядку. Перед выполнением резервного копирования база может быть проверена на наличие ошибок, а уже созданный файл резервной копии может быть упакован в zip архив.

В настоящем учебном пособии все примеры выполнены на SQL СУБД Firebird 2.1 в среде IBExpert.

## 2.5. Правила синтаксиса и основные запросы SQL

При описании языка SQL будут использоваться синтаксические обозначения, представленные в табл. 2.2 [16].

Следует также учесть, что при дальнейшем описании конструкции базовая\_таблица, представление и столбец будут подразумевать под собой имя базовой таблицы, имя представления и имя столбца соответственно. Конструкция <таблица> будет использоваться для обобщения таких видов таблиц, как базовая таблица, представление и производная таблица. В тех случаях, когда потребуется явно обозначить вид таблицы в синтаксисе, будут указываться базовая\_таблица, представление или <производная\_таблица>.

**Таблица 2.2.** Специальные синтаксические обозначения

Обозначение	Описание
::=	Равно по определению
SELECT	Прописные латинские буквы и символы используются для написания конструкций языка SQL и должны (если это специально не оговорено) записываться в точности так, как показано
<условие>	Строчные буквы, заключенные в скобки < >, используются для сокращенного обозначения конструкций, которые при дальнейшем описании синтаксиса будут раскрываться уровнем за уровнем для получения полной детализации (для определенности отдельные слова этих конструкций связываются между собой символом подчеркивания _)
[]	Квадратные скобки означают, что конструкции, заключенные в эти скобки, являются необязательными (т.е. могут быть опущены)
{ }	Фигурные скобки предполагают обязательный выбор некоторой конструкции из списка. Конструкции, заключенные в эти скобки, должны рассматриваться как целые синтаксические единицы, т.е. они позволяют уточнить порядок разбора синтаксических конструкций, заменяя обычные скобки, используемые в синтаксисе SQL

Таблица 2.2. Окончание

...	Многоточие указывает на то, что непосредственно предшествующая ему синтаксическая единица может повторяться один или более раз
	Прямая черта означает наличие выбора из двух или более возможностей (ИЛИ). Например, обозначение ASC DESC указывает, что можно выбрать один из терминов ASC или DESC; когда же один из элементов выбора заключен в квадратные скобки, то это означает, что он выбирается по умолчанию (так, [ASC] DESC означает, что отсутствие всей этой конструкции будет восприниматься как выбор ASC)
*	Используется для обозначения "все". Употребляется в обычном для программирования смысле, т.е. "все случаи, удовлетворяющие определению"
;	Точка с запятой - завершающий элемент SQL-запросов
,	Запятая используется для разделения элементов списков
Пробел	Может вводиться для повышения наглядности между любыми синтаксическими конструкциями предложений SQL

БД Firebird создается и управляется запросами языка SQL. По своей сущности запрос является командой SQL (или, как часто указывается в литературе, оператором SQL), которая передается на сервер. Строго говоря, команда не является запросом, пока она не будет передана серверу. При этом большинство разработчиков используют термины оператор, команда и запрос для обозначения одних и тех же синтаксических конструкций SQL. В данном учебном пособии для обозначения различных команд SQL используется термин *запрос*.

Запросы языка SQL можно разделить на следующие шесть основных категорий:

- язык определения данных (Data Definition Language – DDL) позволяет создавать и изменять структуру объектов БД, например создавать и удалять таблицы;
- язык манипулирования данными (Data Manipulation Language – DML) используется для манипулирования информацией внутри объектов реляционной БД, например добавляет строки в таблицы;
- язык выборки данных (Data Query Language –DQL) включает один запрос SELECT, который вместе со своими многочисленными опциями и предложениями используется для формирования запросов к реляционной БД;
- язык управления доступом к данным (Data Control Language – DCL) позволяет управлять доступом к информации, находящейся внутри БД. Как правило, данный язык используется для создания объектов, связанных с доступом к данным, а также служат для контроля над распределением привилегий между пользователями;
- язык управления транзакциями;



- язык администрирования данных, с помощью команд которого пользователь контролирует выполняемые действия и анализирует операции БД. Он также используется при анализе производительности системы. Не следует путать администрирование данных с администрированием БД, которое представляет собой общее управление БД и подразумевает использование запросов всех уровней.

Отдельно следует отметить процедурный язык Firebird (PSQL), который используется при написании хранимых процедур и триггеров. В настоящем учебном пособии при описании конструкций, доступных для использования только в процедурном языке, будет использоваться термин *оператор*.

Основные категории запросов языка SQL предназначены для выполнения различных функций, включая построение объектов БД и манипулирование ими, начальную загрузку данных в таблицы, обновление и удаление существующей информации, выполнение запросов к БД, управление доступом к ней и ее общее администрирование.

Независимые структуры БД – таблицы, индексы и представления - создаются DDL-запросами. Объекты БД, созданные DDL-запросами, называются метаданными (metadata). На физическом уровне метаданные представляют собой системную БД. Такая БД содержит "данные о данных", т.е. определение других объектов системы. Метаданные хранятся в виде системных таблиц, которые автоматически создаются и изменяются Firebird.

Таким образом, процесс определения данных – это процесс создания, изменения и удаления метаданных.

Запросы DML, DQL, DCL оперируют с существующими данными, хранящимися в структурах БД, определенных запросами DDL.

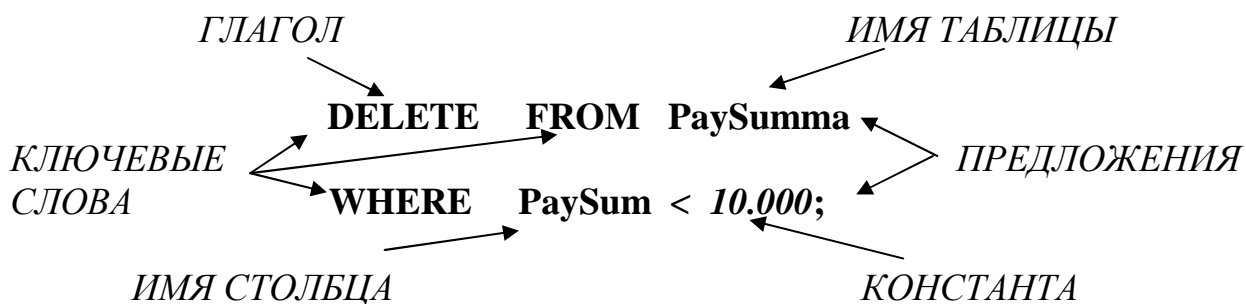
Запросы SQL сообщают СУБД о необходимости выполнить определенное действие. Запросы SQL позволяют:

- создать таблицу;
- читать данные;
- получать итоговые данные;
- добавлять данные;
- удалять данные;
- обновлять (изменять) данные;
- защищать данные.

SQL-запрос состоит из ключевых слов и слов, определяемых пользователем.

На рис. 2.5 приведен примерный формат SQL-запроса.

Ключевые слова являются постоянной частью языка SQL и имеют фиксированное значение. Их следует записывать в точности так, как это установлено, нельзя разбивать на части для переноса с одной строки на другую. Слова, определяемые пользователем, задаются им самим (в соответствии с синтаксическими правилами) и представляют собой идентификаторы или имена различных объектов БД. Слова в запросе размещаются также в соответствии с установленными синтаксическими правилами.



**Рис. 2.5.** Формат SQL-запроса

Каждый запрос начинается с глагола, т.е. ключевого слова, описывающего выполняемое действие, и заканчивается точкой с запятой.

Типичными глаголами являются SELECT (выбрать), CREATE (создать), INSERT (добавить), DELETE (удалить) и COMMIT (завершить).

После глагола следует одно или несколько предложений. Они описывают данные, с которыми работает запрос, или содержат уточняющую информацию о действии, выполняемом запросом. Каждое предложение начинается с ключевого слова, например WHERE (где), FROM (откуда), INTO (куда) и HAVING (имеющий). Одни предложения в запросе могут изменяться, а другие - нет. При этом конкретная структура и содержимое предложения также могут изменяться. Многие предложения содержат имена таблиц или столбцов; некоторые из них могут содержать дополнительные ключевые слова, константы и выражения.

При дальнейшем описании глаголы, с которых начинаются запросы, и ключевые слова (слова, которые в SQL зарезервированы для специального использования и являются частью его синтаксиса) будут записываться заглавными буквами, чтобы отличать их от имен столбцов и таблиц. Но в общем случае синтаксис SQL-запросов не чувствителен к расположению текста по строкам и к регистру символов. Более подробно запросы языка SQL рассмотрим далее. В табл. 2.3 перечислены основные запросы SQL [17].

При выполнении каждый запрос SQL проходит несколько фаз обработки [17]:

- синтаксический разбор, который включает проверку синтаксиса запроса, проверку имен таблиц и столбцов в БД, а также подготовку исходных данных для оптимизатора;
- проверка привилегий пользователя, проверка действительности имен системных каталогов, таблиц и названий полей;
- генерация плана доступа к ресурсам (план доступа - это двоичное представление выполнимого кода по отношению к данным, сохраняемым в БД);
- оптимизация плана доступа, которая включает подстановку действительных имен таблиц и колонок БД в представление, идентификацию возможных вариантов выполнения запроса, определение стоимости выполнения каждого варианта, выбор наилучшего варианта на основе внутренней статистики;

- выполнение запроса.

В настоящее время оптимизатор является составной частью любой промышленной реализации SQL. Работа оптимизатора основана на сборе статистики о выполняемых запросах и выполнении эквивалентных алгебраических преобразований с отношениями БД. Такая статистика сохраняется в системном каталоге БД. Системный каталог является словарем данных для каждой БД и содержит информацию о таблицах, представлениях, индексах, колонках, пользователях и их привилегиях доступа. Каждая БД имеет свой системный каталог, который представляет совокупность предопределенных таблиц БД.

**Таблица 2.3. Основные запросы SQL**

<b>Запрос</b>	<b>Описание</b>
<i>Язык определения данных (DDL)</i>	
CREATE DOMAIN	Определяет новый домен
DROP DOMAIN	Удаляет домен
ALTER DOMAIN	Изменяет домен
CREATE TABLE	Создает в БД новую базовую таблицу
DROP TABLE	Удаляет базовую таблицу из БД
ALTER TABLE	Изменяет структуру существующей базовой таблицы
CREATE VIEW	Добавляет в БД новое представление
DROP VIEW	Удаляет представление из БД
CREATE INDEX	Создает индекс для столбца
DROP INDEX	Удаляет индекс столбца
ALTER INDEX	Изменяет индекс
<i>Язык манипулирования данными (DML)</i>	
INSERT	Добавляет новые строки в таблицы БД
DELETE	Удаляет строки из таблиц БД
UPDATE	Обновляет данные, существующие в таблицах БД
<i>Язык выборки данных (DQL)</i>	
SELECT	Считывает данные из таблиц БД
<i>Язык управления доступом (DCL)</i>	
GRANT	Предоставляет пользователю права доступа
REVOKE	Отменяет права доступа
<i>Язык управления транзакциями</i>	
COMMIT	Завершает текущую транзакцию
ROLLBACK	Отменяет текущую транзакцию
SAVEPOINT	Назначает контрольную точку внутри транзакции
<i>Язык администрирования базы данных</i>	
CREATE DATABASE	Физически создает БД
CONNECT	Подключает к существующей базе данных

## 2.6. Формы использования SQL

Язык SQL может выступать в различных формах [18, 19]. SQL можно использовать как в интерактивном режиме (*Interactive SQL* - ISQL), так и путем внедрения его запросов в программы, написанные на процедурных языках высокого уровня.

Интерактивный SQL позволяет конечному пользователю в интерактивном режиме выполнять SQL-запросы и просматривать результаты их выполнения. Все СУБД предоставляют инструментальные средства для работы с базой данных в интерактивном режиме. Для СУБД Firebird 2.1 "родным" (входящим в комплект установки) инструментом интерактивного SQL является одна из утилит командной строки – isql. Как уже отмечалось, также можно использовать инструменты с графическим интерфейсом (например, IBExpert). Графические инструменты предоставляют возможность как непосредственно вводить текст SQL-запроса в редакторе, так и визуально составлять запросы с помощью построителя (конструктора), что часто значительно упрощает написание запроса.

Применение языка SQL в прикладных программах на практике реализовано двумя различными способами.

1. Внедренные SQL-запросы. Отдельные SQL-запросы внедряются прямо в исходный текст программы и смешиваются с операторами базового языка. Этот подход позволяет создавать программы, обращающиеся непосредственно к базе данных. Специальные программы-предкомпиляторы преобразуют исходный текст с целью замены SQL-запросов соответствующими вызовами подпрограмм СУБД, затем он компилируется и собирается обычным способом.

2. Использование прикладного интерфейса программирования (API – Application Programming Interface), позволяющего реализовывать работу с базой данных через предоставляемый набор функций. API может быть целевым, предоставленным производителем коммерческой СУБД для работы именно с этой базой данных, или межплатформенным, реализующим унифицированные средства доступа к СУБД различных производителей. Конкретный вариант API может предоставлять тот же набор функциональных возможностей, который существует при подключении встроенных операторов, однако при этом устраняется необходимость предкомпилирования исходного текста. Кроме того, некоторые разработчики указывают, что в этом случае используется более понятный интерфейс и созданный программный текст более удобен с точки зрения его сопровождения. Прикладной API включает набор библиотечных функций, предоставляющих программисту разнообразные типы доступа к базе данных, а именно: подключение, выполнение различных SQL-запросов, выборка отдельных строк данных из результирующих наборов данных и т. д.

Внедрение SQL-запросов в текст программы предполагает использование операторов как статического SQL, так и динамического SQL (Dynamic SQL - DSQL) [18].

*Статический SQL* может реализовываться как *встроенный SQL* (Embedded SQL - ESQL) или модульный SQL. Во встроенном SQL можно использовать

переменные основного языка программирования. Запросы статического SQL определены уже в момент компиляции программы.

Что касается запросов статического SQL, то какого-либо изменения после их однократного написания не предполагается. Они могут храниться как в файлах, предназначенных для дальнейшего использования, так и в виде хранимых процедур базы данных, однако программисты не получают всей той гибкости, которую предлагает им динамический SQL. Несмотря на наличие большого числа запросов, доступных конечному пользователю, может случиться так, что ни один из этих "законсервированных" запросов не сможет удовлетворить его текущим потребностям.

*Динамический SQL* (в отличие от статического SQL) позволяет формировать SQL-запросы не на этапе компиляции, а во время выполнения программы. Запросы динамического SQL формируются как текстовые переменные (например, `Abon:='SELECT * FROM ABONENT'`;). Для динамического формирования оператора можно выполнять последовательное объединение строк.

Динамический SQL дает возможность программисту или конечному пользователю создавать операторы во время выполнения приложения и передавать их базе данных, которая после выполнения этих операторов помещает выходные данные в переменные программы. Динамический SQL часто используется инструментальными средствами, предназначенными для построения заранее незапланированных запросов, позволяющих оперативно формировать тот или иной SQL-запрос в зависимости от особых требований, возникших в конкретной ситуации. После настройки SQL-запроса в соответствии с потребностями пользователя он направляется серверу баз данных для проверки на наличие синтаксических ошибок и необходимых для его выполнения привилегий, после чего происходит его компиляция и выполнение.

Стандартный набор запросов SQL дополняется так называемым **процедурным языком Firebird (PSQL)**, который предполагает расширение SQL средствами программирования. Это не встроенный SQL, предполагающий включение конструкций SQL в текст программы, написанной на полноценном языке программирования. Процедурный язык – это язык, который интерпретирует сама СУБД. На процедурном языке создаются триггеры и хранимые процедуры (ХП). ХП похожи на процедуры языков высокого уровня, для получения исходных данных и для вывода результатов ХП могут иметь входные и выходные параметры соответственно. Триггеры выполняются автоматически при изменении данных в БД, они не имеют параметров, но могут использовать ряд специальных контекстных переменных, не доступных в ХП.

## **2.7. Имена объектов в SQL. Константы, отсутствующие данные**

У каждого объекта в БД есть свое уникальное имя. Имена используются в SQL-запросах для указания объектов БД, над которыми запрос должен выполнить действие. Имена имеются у баз данных, таблиц, столбцов,

представлений, курсоров и пользователей. Часто в SQL поддерживаются также именованные триггеры, хранимые процедуры, именованные отношения «первичный ключ – внешний ключ» и формы для ввода данных. В SQL имена должны содержать от 1 до 31 символа [прописные и/или строчные буквы латинского алфавита, цифры, символ подчеркивания (\_), знак доллара (\$)], начинаться с буквы и не содержать пробелы или специальные символы пунктуации:

<идентификатор> ::= буква { буква | цифра } ...

В БД диалекта 3 Firebird поддерживает соглашение ANSI SQL об идентификаторах с разделителями [18]. Для использования зарезервированных слов, символов, чувствительных к регистру, или пробелов в имени объекта следует заключить это имя в двойные кавычки. Такой идентификатор – *идентификатор с разделителем*. Данные идентификаторы должны быть представлены с двойными кавычками во всех типах запросов SQL. Идентификаторы с разделителями были введены для совместимости со стандартами, но если нет серьезных оснований их использовать, то рекомендуется использовать обычные идентификаторы.

Если в запросе указано имя таблицы, предполагается, что происходит обращение к одной из таблиц подключенной БД. К конкретной таблице БД можно обращаться, только будучи ее владельцем или имея соответствующее разрешение, данное владельцем. Владельцем БД или какой-то ее таблицы считается пользователь, который создает ее.

Если в запросе задается имя столбца, SQL сам определяет, в какой из указанных в этом же операторе таблиц содержится данный столбец. Однако если в запрос требуется включить два столбца из различных таблиц, но с одинаковыми именами, необходимо указать полные имена столбцов. Полное имя столбца состоит из имени таблицы, содержащей столбец, и имени столбца (простого имени), разделенных точкой. Например, полное имя столбца AccountCD (номер лицевого счета абонента) из таблицы Abonent имеет следующий вид: Abonent.AccountCD.

В SQL-запросе можно конкретно указывать значения чисел, строк или даты. Для этого определен формат числовых и строковых констант (литералов), представляющих конкретные значения данных:

- целые и десятичные константы в SQL-запросе представляются в виде обычных десятичных чисел с необязательным знаком плюс (+) или минус (-) перед ними;
- константы с плавающей запятой определяются с помощью символа E и имеют такой же формат, как и в большинстве языков программирования;
- строковые константы заключаются в одинарные кавычки, например: 'FIREBIRD'. Если необходимо включить в строковую константу одинарную кавычку, то следует ввести две одинарные кавычки, например: 'IT'S MY LIFE'. В таблицу БД это будет помещено как IT'S MY LIFE;

- календарные даты в стандарте представляются в виде строковых констант. Например, 1 сентября 1999 года должно быть представлено как один из следующих вариантов: '09/01/1999', '09:01:1999', '01.09.1999', '01-SEP-1999' или '1999-09-01'.

В БД некоторые данные могут отсутствовать из-за того, что они неизвестны на данный момент времени или не существуют по своей природе.

Если не вводить эти данные, то в БД будут неопределенные значения, что недопустимо. Поэтому SQL поддерживает обработку отсутствующих данных с помощью понятия "отсутствующее значение". Оно показывает, что в конкретной строке определенный элемент данных отсутствует или столбец вообще не подходит для этой строки. Говорят, что значение такого элемента данных равно NULL. Однако NULL не является значением данных, как 0, или 457, или 'Поставщик'. Напротив, это признак, показывающий, что точное значение данных неизвестно или отсутствует.

Во многих ситуациях значения NULL требуют от СУБД отдельной обработки.

## 2.8. Выражения

**Выражения** представляют собой комбинацию идентификаторов, функций, операций, констант и других *объектов*. *Выражение* может быть использовано в качестве аргумента в хранимых процедурах или запросах.

*Выражение* состоит из *операндов* (собственно *данных*) и *операций*. В качестве *операндов* могут выступать константы, *переменные*, имена столбцов, функции, подзапросы.

*Операции* – это определенный вид действий над элементами данных, после выполнения которых получается в качестве результата новое значение. Элементы данных, используемые при выполнении операций, называются *операндами* или *аргументами*. Операции представляются в виде специальных символов или ключевых слов.

В Firebird 2.1 предусмотрено несколько категорий операций (табл. 2.4) [16].

В SQL определено понятие **скалярного выражения** [9]. Скалярное выражение - это выражение, вырабатывающее результат некоторого типа, специфицированного в стандарте. *Скалярные выражения* являются основой языка SQL, поскольку, хотя это реляционный язык, все условия, элементы списков выборки и т. д. базируются именно на *скалярных выражениях*. В стандарте SQL имеется несколько разновидностей *скалярных выражений*. К числу наиболее важных разновидностей относятся *численные выражения* (это выражения, значения которых относятся к числовому типу данных); *строковые выражения* (это выражения, значениями которых являются символьные строки); *выражения со значениями «даты-времени»* (выражения, вырабатывающие значения типа «дата-время»); логические выражения.

**Таблица 2.4.** Категории операций

<b>Категория операции</b>	<b>Описание</b>
Унарная операция	Выполняется лишь над одним выражением любого типа данных из категории числовых типов данных
Строковая операция	Операция конкатенации (  ), которая соединяет строки, являющиеся ее операндами
Арифметические операции	+, -, *, /
Логические операции	Проверяют истинность условий AND, OR, NOT
Операции сравнения	Сравнивают значение с другим значением или выражением (=, <>, <, >, <=, >=)
Операция выбора	Возвращает различные результаты в зависимости от определенных условий (CASE)
Специальные предикаты	Расширяют возможности операций сравнения (ANY, ALL, BETWEEN, DISTINCT FROM, EXIST, SINGULAR, IN, IS NULL, LIKE, CONTAINING, STARTING WITH)

Основой логического выражения являются предикаты. Предикат позволяет специфицировать условие, результатом вычисления которого может быть TRUE, FALSE или UNKNOWN. В SQL ложный и неопределенный результаты объединяются и расцениваются как ложь. Решения принимаются в соответствии с результатом вычисления предиката – истина или ложь. Можно сказать, что предикат – это логическая функция, принимающая значения «истина» или «ложь». Синтаксическими элементами, проверяющими истинность, являются:

- в языке DDL: CHECK для проверки условий достоверности данных;
- в языке DQL: WHERE (для условий поиска), HAVING (для условий выбора групп), ON (для условий соединения) и случаи проверки множества условий: CASE, COALESCE, NULLIF, IF;
- в языке PSQL: IF (универсальная проверка истина/ложь), WHILE (для проверки условий цикла) и WHEN (для проверки кодов исключения).

Часто условия являются не простыми предикатами, а группой нескольких предикатов (связанных логическими операциями AND или OR), каждый из которых при вычислении делает вклад в вычисление общей истинности.

Для построения всех этих видов выражений могут использоваться константы, переменные, имена столбцов, функции и подзапросы, возвращающие результат соответствующего типа в зависимости от разновидности скалярного выражения. Во все эти виды выражений могут входить CASE-выражения (или выражения с переключателем), которые представляют собой либо операцию CASE, либо функции вывода (выбора вариантов). Более подробно CASE-выражения будут рассмотрены далее.



Простым примером численного выражения является  $2*2$ . Следующее выражение является более сложным и использует функцию и строковую операцию ||:

```
'Год подачи ремонтной заявки' || EXTRACT (YEAR From IncomingDate).
```

## 2.9. Типы данных

Типы данных определяют столбцы таблицы БД в следующих случаях:

- при определении (создании) таблицы запросом CREATE TABLE;
- при определении домена запросом CREATE DOMAIN;
- при добавлении нового столбца запросом ALTER TABLE.

В SQL используются основные типы данных, форматы которых могут несколько различаться для разных СУБД. Общий вид определения типа данных выглядит следующим образом:

```
<тип_данных>::=  
{ {SMALLINT | INTEGER | BIGINT | FLOAT | DOUBLE PRECISION}  
  | {DECIMAL | NUMERIC} [( PRECISION [, SCALE])] }  
  | {DATE | TIME | TIMESTAMP}  
  | {CHAR | VARCHAR} | BLOB }.
```

В табл. 2.5 приведено описание типов данных SQL, используемых в СУБД Firebird [18].

Символьные типы данных представлены типами CHAR и VARCHAR.

Тип CHAR определяет строки фиксированной длины. Если при вводе строка символов меньше объявленного значения длины n, то недостающие символы заполняются пробелами, а в случае превышения – усекаются.

Тип VARCHAR представляет строки переменной длины. В отличие от типа CHAR при сохранении в БД недостающие символы не заполняются пробелами. При чтении считываются только введенные символы, а затем дополняются пробелами.

Символьные типы фиксированной длины не могут превышать 32767 байт абсолютной длины; для типов переменной длины этот предел уменьшается на два байта, которые при сохранении строки содержат счетчик символов [18].

Числовые типы данных представлены типами SMALLINT, INTEGER и BIGINT, NUMERIC и DECIMAL, DOUBLE PRECISION и FLOAT.

Целые числовые типы (согласно стандарту SQL99) – это типы SMALLINT, INTEGER, BIGINT. Они представляют собой 16-битный, 32-битный и 64-битный типы со знаком соответственно. Если величина SMALLINT (5 значащих цифр) должна быть меньше или равна величине INTEGER (10 значащих цифр), то величина BIGINT (19 значащих цифр) должна быть больше или равна величине INTEGER.

**Таблица 2.5.** Типы данных SQL, используемых в СУБД Firebird

Тип данных	Размер	Диапазон/Точность	Описание
CHAR(n)	n символов	От 1 до 32767 байт	Строки символов постоянной длины $0 < n \leq 32767$
VARCHAR(n)	n символов	От 1 до 32765 байт	Строки символов переменной длины $0 < n \leq 32765$
SMALLINT	16 бит	От -32768 до 32767	Знаковые целые числа (короткие)
INTEGER	32 бита	От -2147483648 до 2147483647	Знаковые целые числа
BIGINT	64 бита	От -9223372036854775808 до 9223372036854775807 (от $-2^{63}$ до $2^{63}-1$ )	Знаковые целые числа (большие)
NUMERIC(p,s)	Переменный	P=1 до 15; s=1 до 15	Масштабируемые десятичные числа; p-точность; s-число знаков после запятой (масштаб); $p \geq s$
DECIMAL(p,s)			То же, что и NUMERIC(p,s)
DOUBLE PRECISION	64 бита	От $1.7 \cdot 10^{-308}$ до $1.7 \cdot 10^{308}$	Числа с плавающей запятой высокой точности; 15 разрядов после запятой
FLOAT	32 бита	От $3.4 \cdot 10^{-38}$ до $3.4 \cdot 10^{38}$	Числа с плавающей запятой; 7 разрядов после запятой
DATE	32 бита	От 1 января 100 года до 27 февраля 32768 года	Календарная дата
TIME	32 бита	От 00:00 до 23:59:59	Время дня
TIMESTAMP	2*32 бита	Дата от 1 января 100 года до 27 февраля 32768 года и время от 00:00 до 23:59:59	Календарная дата и время
BLOB	Переменный	Нет	Динамически изменяемый; для хранения больших объемов данных

Десятичными числовыми типами с фиксированной точкой являются типы NUMERIC и DECIMAL.

При определении типа NUMERIC или DECIMAL Firebird производит преобразование типов тремя способами:

- определенные без p и s - всегда сохраняются как INTEGER;
- определенные с p, но без s – в зависимости от спецификации p, сохраняются как SMALLINT, INTEGER или BIGINT;
- определенные с p и s – в зависимости от спецификации p, сохраняются как SMALLINT, INTEGER или BIGINT.

В табл. 2.6 содержатся сведения о том, как Firebird сохраняет десятичные типы данных NUMERIC (или DECIMAL).

**Таблица 2.6. Преобразование типов данных**

<b>Определяемый тип</b>	<b>Тип, сохраняемый Firebird</b>
NUMERIC	INTEGER
NUMERIC(4)	SMALLINT
NUMERIC(9)	INTEGER
NUMERIC(10)	BIGINT
NUMERIC(4,2)	SMALLINT
NUMERIC(9,3)	INTEGER
NUMERIC(10,4)	BIGINT
DECIMAL	INTEGER
DECIMAL(4)	INTEGER
DECIMAL(9)	INTEGER
DECIMAL(10)	BIGINT
DECIMAL(4,2)	INTEGER
DECIMAL(9,3)	INTEGER
DECIMAL(10,4)	BIGINT

Приведенные в табл. 2.6 данные по преобразованию десятичных типов следует понимать следующим образом. Если, например, определить десятичный тип с  $p \leq 9$  и  $s \leq 3$ , то тип NUMERIC(9,3) будет преобразован в INTEGER, но, приняв определение для  $p \geq 10$  и  $s \geq 4$ , будет получен тип BIGINT.

Числовыми типами с плавающей точкой являются типы DOUBLE PRECISION и FLOAT. В этих типах положение десятичной точки не зафиксировано. Столбцы с типом DOUBLE PRECISION или FLOAT следует определять, когда нужно хранить числа с изменяющимся масштабом.

Типы данных даты и времени представлены типами DATE, TIME и TIMESTAMP.

Формат типа DATE задается в виде 'ДД.ММ.ГГГГ', 'ММ/ДД/ГГГГ', 'ГГГГ-ММ-ДД', 'ММ:ДД:ГГГГ' или 'ДД-МММ-ГГГГ'. При этом ММ – две цифры месяца (МММ – первые три буквы в английском названии месяца), ДД – две цифры дня, ГГГГ – четыре цифры года.

Для формата 'ДД-МММ-ГГГГ' в качестве МММ употребляются первые три буквы латинского названия месяца (табл. 2.7).

Например, дату 17 декабря 2001 года можно записать как один из следующих вариантов: '17.12.2001', '12/17/2001', '2001-12-17', '12:17:2001' или '17-DEC-2001'.

Тип данных TIME включает время. Чтобы задать время, необходимо использовать описание 'ЧЧ/МН/СК' или 'ЧЧ:МН:СК'. При этом ЧЧ – две цифры часов, МН – две цифры минут, СК – две цифры секунд.

Чтобы задать дату и время вместе, необходимо использовать тип TIMESTAMP с описанием 'ММ/ДД/ГГГГ/ЧЧ/МН/СК' или 'ДД.ММ.ГГГГ ЧЧ:МН:СК'.

**Таблица 2.7.** Сокращения названий месяцев

<b>Название месяца</b>	<b>Используемое обозначение</b>
Январь	JAN
Февраль	FEB
Март	MAR
Апрель	APR
Май	MAY
Июнь	JUN
Июль	JUL
Август	AUG
Сентябрь	SEP
Октябрь	OCT
Ноябрь	NOV
Декабрь	DEC

Если в таблице БД или в нескольких таблицах присутствуют столбцы, обладающие одними и теми же характеристиками, можно описать тип такого столбца и его поведение через домен, а затем поставить в соответствие каждому из одинаковых столбцов имя домена. Домен определяет все потенциальные значения, которые могут быть присвоены атрибуту.

Получить список всех типов данных можно из системной таблицы RDB\$TYPES с помощью следующего запроса:

```
SELECT * FROM RDB$TYPES;
```

В языке SQL обеспечивается возможность использования в различных операциях не только значений тех типов, для которых predetermined операция, но и значений типов, неявным или явным образом приводимых к требуемому типу. Кроме неявного преобразования типов данных (например, преобразования десятичных типов – табл. 2.6), существует универсальная функция CAST, с помощью которой значения одного типа преобразовываются в значения другого типа, если такие изменения вообще возможны. Подробнее данная функция будет рассмотрена далее.

Ориентированный на работу с таблицами, SQL не имеет достаточно средств для создания сложных прикладных программ. Поэтому в разных СУБД он либо используется вместе с языками программирования высокого уровня (Си, Паскаль и др.), либо включен в состав команд специально разработанного языка СУБД (язык систем dBASE, R:BAE и т.п.). Унификация полных языков современных профессиональных СУБД достигается за счет внедрения объектно-ориентированного языка четвертого поколения 4GL. Он позволяет организовывать циклы, условные предложения, меню, экранные формы, сложные запросы к базам данных с интерфейсом, ориентированным как на алфавитно-цифровые терминалы, так и на оконный графический интерфейс.

## Контрольные вопросы

1. Какие основные объекты существуют в БД?
2. Какие функции выполняет ядро базы данных?
3. Какие функции выполняет язык SQL?
4. Какие достоинства языка SQL принесли ему успех?
5. Охарактеризуйте основные файлы, входящие в состав сервера Firebird, и опишите, как работает сервер.
6. Какие существуют утилиты для администрирования БД Firebird?
7. Какие существуют категории запросов языка SQL?
8. Какие фазы обработки проходит каждый SQL-запрос при выполнении?
9. В каких формах может использоваться язык SQL?
10. Какие существуют правила составления имен объектов в SQL, какие константы могут использоваться?
11. Что представляет собой выражение в языке SQL? Какие разновидности выражений существуют?
12. Какие типы данных используются в СУБД Firebird?

## 3. Язык выборки данных

В этой главе приводятся возможности SQL по *выборке* данных из БД. Это самая большая глава из представленных в данном учебном пособии, потому что именно для организации выборки информации из БД SQL используется наиболее интенсивно подавляющим большинством пользователей.

Начиная с самых простых возможностей и последовательно переходя ко все более сложным конструкциям, в данной главе рассматриваются практически все указанные в стандарте SQL средства выборки данных. Кроме того, представлены некоторые дополнительные возможности, реализованные в СУБД Firebird.

### 3.1. Синтаксис запроса SELECT

Для выборки данных из БД используется запрос SELECT. Он позволяет производить выборку требуемых данных из таблиц и преобразовывать к нужному виду полученные результаты. В общем случае результатом реализации запроса SELECT является другая таблица, которую будем называть таблицей результатов запроса (ТРЗ). К этой новой (рабочей) таблице может быть снова применен запрос SELECT и т.д., т.е. такие операции могут быть вложены друг в друга.

Запрос SELECT может использоваться как:

- самостоятельная команда на получение и вывод строк таблицы, сформированной из столбцов и строк одной или нескольких таблиц (представлений);
- элемент SELECT, WHERE- или HAVING-условия (сокращенный вариант предложения, называемый "вложенный запрос");

- запрос на формирование данных представления в команде CREATE VIEW;
- средство выборки информации, необходимой для модификации данных в других таблицах (многострочные запросы DML);
- средство присвоения глобальным переменным значений из строк сформированной таблицы (INTO-фраза).

Полный синтаксис запроса SELECT имеет следующий вид [19, 20]:

```
[WITH [RECURSIVE]
  имя_производной_таблицы1 [(<список_столбцов>)]
  AS (<табличный_подзапрос> )
  [, имя_производной_таблицы2 [(<список_столбцов>)]
  AS (<табличный_подзапрос> )...]
SELECT [DISTINCT | ALL]
      [FIRST m] [SKIP n]
      { * | <возвращаемый_элемент1> [[AS] псевдоним_элемента1]
      [, <возвращаемый_элемент2> [[AS] псевдоним_элемента2] ]...}
FROM { <таблица1> [псевдоним1] [, <таблица2> [псевдоним2]...
      | <таблица1> [псевдоним1] <тип_соединения1> <таблица2> [псевдоним2]
      [{ON<условие_соединения1> | USING (<список_столбцов>)}]}
      [<тип_соединения2> <таблица3> [псевдоним3]
      [{ON<условие_соединения2> | USING (<список_столбцов>)}]}]...}
[WHERE <условие_поиска>]
[GROUP BY <элемент_группировки1> [, <элемент_группировки2>]...]
[HAVING <условие_поиска>]
[PLAN <список_пунктов_плана>]
[ORDER BY <элемент_сортировки1> [, <элемент_сортировки 2>]...]
[ROWS k [TO r ]],
```

где

`<список_столбцов>`:: = столбец1 [, столбец2 ...];

`<возвращаемый_элемент>` :: =  
 { [`<таблица>`].\* | [`<таблица>`].столбец | константа | `<выражение>`  
 | (`<скалярный_подзапрос>` ) };

`<таблица>`:: =  
 { базовая\_таблица | представление | имя\_производной\_таблицы  
 | `<производная_таблица>`};

`<производная_таблица>` ::=  
 (`<табличный_подзапрос>`) [[AS] псевдоним] [(`<список_столбцов>`)];

`<тип_соединения>`::=  
 {CROSS JOIN  
 | [NATURAL] [{INNER | {LEFT | RIGHT | FULL} [OUTER]}] JOIN};

`<условие_соединения>`::=  
 {`<таблица1>`.столбец `<операция_сравнения>` `<таблица2>`.столбец};

<операция\_сравнения> ::= {= | < | > | <= | >= | <> };

<условие\_поиска> ::= [NOT] <условие\_поиска1>  
[[AND|OR][NOT] <условие\_поиска2>]....,

где

<условие\_поиска> ::=

{ <значение> <операция\_сравнения> {<значение1>  
| (<скалярный\_подзапрос> )  
| {ANY| ALL} (<подзапрос\_столбца>)}  
| <значение> [NOT] BETWEEN <значение1> AND <значение2>  
| <значение> [NOT] LIKE 'шаблон' [ESCAPE 'символ\_пропуска']  
| <значение> [NOT] CONTAINING <значение1>  
| <значение> [NOT] STARTING WITH <значение1>  
| <значение> [NOT] IN ({<значение1> [, <значение2> ...] | <подзапрос\_столбца>)}  
| <значение> IS [NOT] NULL  
| <значение> IS [NOT] DISTINCT FROM <значение1>  
| EXISTS (<табличный\_подзапрос>)  
| SINGULAR (<табличный\_подзапрос>)};

<значение> ::= {[<таблица>.] столбец | константа | <выражение> | функция};

<элемент\_сортировки> ::= { [<таблица>.] столбец  
| порядковый\_номер\_столбца  
| псевдоним\_столбца  
| <выражение> }

[ASC[ENDING] | DESC [ENDING]]  
[NULLS FIRST | NULLS LAST]}...;

<элемент\_группировки> := { [<таблица>.] столбец  
| порядковый\_номер\_столбца  
| псевдоним\_столбца  
| <выражение> }.

Конкретные особенности использования приведенных выше конструкций поясняются далее в процессе изучения запроса SELECT.

## 3.2. Запросы к одной таблице

Применительно к однотабличным запросам SELECT имеет следующий формат:

```
SELECT [DISTINCT | ALL]
      [FIRST m] [SKIP n]
      { * | <возвращаемый_элемент1> [AS псевдоним_элемента1]
        [, <возвращаемый_элемент2> [AS псевдоним_элемента2] ] ... }
FROM {базовая_таблица | представление} [псевдоним]
[WHERE <условие_поиска>]
[GROUP BY <элемент_группировки1> [, <элемент_группировки2>] ...]
```

[HAVING <условие\_поиска>  
[PLAN <список\_пунктов\_плана>  
[ORDER BY <элемент\_сортировки1> [, <элемент\_сортировки 2>]...]  
[ROWS k [TO r ]].

Обработка элементов запроса SELECT выполняется в следующей последовательности:

- 1) FROM – определяются имена используемых объектов;
- 2) WHERE – выполняется *фильтрация строк* объекта в соответствии с заданными условиями;
- 3) GROUP BY – образуются *группы строк*, имеющих одно и то же значение в указанном элементе (столбце);
- 4) HAVING – фильтруются группы строк объекта в соответствии с указанным условием;
- 5) SELECT – устанавливается, какие элементы должны присутствовать в выходных данных;
- 6) ORDER BY – определяется упорядоченность результатов выполнения запроса.

Порядок предложений и фраз в запросе SELECT не может быть изменен. Только два предложения – SELECT и FROM являются обязательными, все остальные могут быть опущены. Существует множество вариантов записи данного запроса, что иллюстрируется приведенными ниже примерами.

Ниже перечислены функции каждого из предложений.

В предложении SELECT указывается список столбцов ТРЗ, которые должны быть возвращены запросом SELECT. Возвращаемые элементы могут содержать значения, считываемые из столбцов таблицы БД, или значения, вычисляемые во время выполнения запроса.

Конструкция DISTINCT | ALL определяет, что делать с повторяющимися строками результата. При использовании конструкции ALL возвращаются все строки, удовлетворяющие условиям запроса (этот режим используется по умолчанию). При использовании конструкции DISTINCT возвращаются только неповторяющиеся строки.

Конструкция FIRST...SKIP служит для ограничения возвращаемых запросом строк.

В предложении FROM указывается список объектов БД, которые содержат данные, считываемые запросом.

Предложение WHERE показывает, что в результаты запроса следует включать только некоторые строки. Для отбора строк, включаемых в результаты запроса, используется условие поиска.

Предложение GROUP BY позволяет создать итоговый запрос, который вначале группирует строки таблицы по определенному признаку, а затем включает в результаты запроса одну итоговую строку для каждой группы.

Предложение HAVING показывает, что в ТРЗ следует включать только некоторые из групп, созданных с помощью предложения GROUP BY. В этом



предложении, как и в предложении WHERE, для отбора включаемых групп используется условие поиска.

Предложение PLAN служит для определения пользователем собственных способов выполнения запросов. Обычно составлением плана запроса занимается *оптимизатор*. Оптимизатор анализирует возможные пути выполнения запроса, определяет «стоимость» выполнения каждого варианта и выбирает наилучший (наиболее быстрый) вариант. В основном пользователь не использует пункт PLAN, и СУБД сама определяет план по умолчанию.

Предложение ORDER BY сортирует результаты запроса на основании данных, содержащихся в одном или нескольких столбцах ТРЗ. Если это предложение не указано, результаты запроса не будут отсортированы.

Предложение ROWS...TO, как и FIRST...SKIP, служит для ограничения возвращаемых запросом строк, но имеет более широкое применение (может использоваться при объединении результатов нескольких запросов, в любых видах подзапроса, в UPDATE и DELETE).

Рассмотрим подробнее описанные конструкции.

### 3.2.1. Предложения SELECT и FROM

В предложении SELECT, с которого начинается запрос SELECT, необходимо указать элементы данных, которые будут возвращены в результате запроса. Эти элементы составляют столбцы ТРЗ и задаются в виде списка возвращаемых элементов, разделенных запятыми.

Синтаксис возвращаемых элементов применительно к однотабличным запросам имеет следующий вид:

```
<возвращаемый_элемент> ::=  
{ * | столбец | константа | <выражение> }.
```

Из приведенного синтаксиса следует, что возвращаемый элемент может представлять собой:

- \*, означающую вывод всех столбцов указанной таблицы;
- имя столбца, идентифицирующее один из столбцов, содержащихся в таблице, указанной в предложении FROM. Когда в качестве возвращаемого элемента указывается имя столбца таблицы БД, происходит выбор значения этого столбца для каждой из строк таблицы и помещение его в соответствующую строку ТРЗ;
- константу, показывающую, что в каждой строке результатов запроса должно содержаться одно и то же значение;
- выражение, показывающее, что необходимо вычислить значение, помещаемое в результат запроса, по формуле, определенной в выражении. Выражения, как уже отмечалось выше, представляют собой комбинацию идентификаторов, функций, операций, констант. Здесь же могут использоваться CASE-выражения;
- простой подзапрос (его использование в качестве возвращаемого элемента будет рассмотрено при изучении вложенных запросов).

Столбцы в таблице результатов располагаются в том порядке, в котором они указаны в списке возвращаемых элементов.

При использовании символа звездочки (\*) в качестве списка возвращаемых элементов из исходного объекта будут прочитаны все столбцы. В случае если выборка производится из нескольких таблиц (многотабличные запросы будут рассмотрены позднее), то перед символом звездочки может указываться имя таблицы. Если не используется символ \*, то для каждого из возвращаемых элементов из списка в ТРЗ будет создан один столбец.

Например, чтобы получить значения *всех* столбцов из таблицы Abonent, необходимо выполнить следующий запрос:

```
SELECT * FROM Abonent;
```

Результат выполнения запроса представлен на рис. 3.1.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
005488	3	4	1	АКСЕНОВ С.А.	556893
115705	3	1	82	МИЩЕНКО Е.В.	769975
015527	3	1	65	КОНЮХОВ В.С.	761699
443690	7	5	1	ТУЛУПОВА М.И.	214833
136159	7	39	1	СВИРИНА З.А.	350003
443069	4	51	55	СТАРОДУБЦЕВ Е.В.	683014
136160	4	9	15	ШМАКОВ С.В.	982222
126112	4	7	11	МАРКОВА В.П.	683301
136169	4	7	13	ДЕНИСОВА Е.К.	680305
080613	8	35	11	ЛУКАШИНА Р.М.	254417
080047	8	39	36	ШУБИНА Т.П.	257842
080270	6	35	6	ТИМОШКИНА Н.Г.	321002

**Рис. 3.1.** Результат выполнения запроса на выборку всех данных

На логическом уровне запрос выполняется путем построчного просмотра таблицы, указанной в предложении FROM. Для каждой строки таблицы берутся значения столбцов, входящих в список возвращаемых элементов, и создается одна строка ТРЗ. Таким образом, таблица результатов простого запроса на чтение, подобного приведенному выше, содержит одну строку данных для каждой строки исходной таблицы БД.

В общем случае предложение FROM состоит из ключевого слова FROM, за которым следует список спецификаторов объектов БД, разделенных запятыми. Каждый спецификатор идентифицирует объект БД (базовая таблица, представление), содержащий данные, которые считывает запрос. Такие объекты называются исходными объектами запроса (и запроса SELECT), поскольку все данные, содержащиеся в ТРЗ, берутся из них. В случае однотабличных запросов в предложении FROM указывается только один исходный объект.

Рассмотрим пример запроса, когда в качестве возвращаемых элементов перечислены имена столбцов базовой таблицы.

Пусть активной является учебная БД. Чтобы вывести номера лицевого счетов, фамилии и телефоны всех абонентов, необходимо выполнить следующий запрос к таблице Abonent:

```
SELECT AccountCD, Fio, Phone FROM Abonent;
```

Здесь в качестве возвращаемых элементов используются имена столбцов AccountCD, Fio, Phone базовой таблицы Abonent.

Имя таблицы можно переопределять, чтобы для ссылок на нее использовать короткий (чаще всего ограничивающийся одной буквой) *псевдоним* (alias), например в следующем виде:

```
SELECT A.AccountCD, A.Fio, A.Phone FROM Abonent A;
```

Результат выполнения двух предыдущих запросов представлен на рис. 3.2.

ACCOUNTCD	FIO	PHONE
005488	АКСЕНОВ С.А.	556893
115705	МИЩЕНКО Е.В.	769975
015527	КОНЮХОВ В.С.	761699
443690	ТУЛУПОВА М.И.	214833
136159	СВИРИНА З.А.	350003
443069	СТАРОДУБЦЕВ Е.В.	683014
136160	ШМАКОВ С.В.	982222
126112	МАРКОВА В.П.	683301
136169	ДЕНИСОВА Е.К.	680305
080613	ЛУКАШИНА Р.М.	254417
080047	ШУБИНА Т.П.	257842
080270	ТИМОШКИНА Н.Г.	321002

**Рис. 3.2.** Результат выполнения запроса к таблице Abonent

Следует учесть, что когда объявлен псевдоним, то нельзя использовать имя соответствующей таблицы в списке возвращаемых столбцов запроса SELECT.

Так, следующий запрос

```
SELECT Abonent.AccountCD, Abonent.Fio, Abonent.Phone
FROM Abonent A;
```

будет выдавать сообщение об ошибке.

В результаты запроса можно включать *константы*, например в следующем виде:

```
SELECT 'Абонент', Fio, 'Номер телефона', Phone FROM Abonent;
```

Результат выполнения запроса представлен на рис. 3.3.

F_1	FIO	F_2	PHONE
Абонент	АКСЕНОВ С.А.	Номер телефона	556893
Абонент	МИЩЕНКО Е.В.	Номер телефона	769975
Абонент	КОНЮХОВ В.С.	Номер телефона	761699
Абонент	ТУЛУПОВА М.И.	Номер телефона	214833
Абонент	СВИРИНА З.А.	Номер телефона	350003
Абонент	СТАРОДУБЦЕВ Е.В.	Номер телефона	683014
Абонент	ШМАКОВ С.В.	Номер телефона	982222
Абонент	МАРКОВА В.П.	Номер телефона	683301
Абонент	ДЕНИСОВА Е.К.	Номер телефона	680305
Абонент	ЛУКАШИНА Р.М.	Номер телефона	254417
Абонент	ШУБИНА Т.П.	Номер телефона	257842
Абонент	ТИМОШКИНА Н.Г.	Номер телефона	321002

**Рис. 3.3.** Результат выполнения запроса с константными выражениями

Как следует из результата предыдущего запроса, СУБД Firebird определяет по умолчанию системные имена для столбцов ТРЗ, содержащих константы.

*Повторяющиеся строки* из таблицы результатов запроса можно удалить, если в запросе SELECT перед списком возвращаемых элементов указать ключевое слово DISTINCT (различный). Например, чтобы выбрать из таблицы Abonent различные значения кодов улиц, необходимо использовать следующий запрос:

```
SELECT DISTINCT StreetCD FROM Abonent;
```

Результат выполнения запроса представлен на рис. 3.4.

STREETCD
3
4
6
7
8

**Рис. 3.4.** Результат запроса на исключение повторяющихся строк

Этот запрос выполняется следующим образом. Вначале генерируются все строки результатов (двенадцать строк – по числу строк в таблице Abonent), а затем удаляются те из них, коды улиц в которых в точности повторяют другие. Ключевое слово DISTINCT можно указывать независимо от содержимого списка возвращаемых элементов запроса SELECT. Если ключевое слово DISTINCT не указано, то повторяющиеся строки не удаляются.

Кроме столбцов, значения которых считываются непосредственно из БД, SQL-запрос на чтение может содержать *вычисляемые столбцы*, значения которых определяются на основании значений данных, хранящихся в БД. Чтобы получить вычисляемый столбец, в списке возвращаемых элементов необходимо указать выражение. Выражения могут включать в себя операции сложения, вычитания, умножения, деления и операцию конкатенации (склеивания) строк, которая обозначается, как две вертикальные линии "||". В выражениях можно также использовать скобки. Таким образом, выражение представляет собой следующую конструкцию:

```
<выражение> ::=  
{ [ [+ ] | - ] { столбец | константа | функция } [ + | - | * | / ] [ || ] }....
```

Например, если требуется вывести в столбце AccountCDRyazan значение лицевого счёта абонента, а в столбце с именем FioPhone – фамилию и номер телефона абонента с добавлением символов 8-4912 в начало номера телефона каждого лицевого счёта, то можно построить следующий запрос:

```
SELECT AccountCD AS AccountCDRyazan,  
       (Fio || ' имеет телефон ' || '8-4912-' || Phone) AS FioPhone  
FROM Abonent;
```

В этом примере использованы *псевдонимы столбцов*, которые задаются с помощью зарезервированного слова AS (как), указываемого после возвращаемого элемента и определяющего имя, под которым столбец будет представлен в ТРЗ.

Результат выполнения запроса представлен на рис. 3.5.

<b>ACCOUNTCDRYAZAN</b>	<b>FIOPHONE</b>
005488	АКСЕНОВ С.А. имеет телефон 8-4912-556893
115705	МИЩЕНКО Е.В. имеет телефон 8-4912-769975
015527	КОНЮХОВ В.С. имеет телефон 8-4912-761699
443690	ТУЛУПОВА М.И. имеет телефон 8-4912-214833
136159	СВИРИНА З.А. имеет телефон 8-4912-350003
443069	СТАРОДУБЦЕВ Е.В. имеет телефон 8-4912-683014
136160	ШМАКОВ С.В. имеет телефон 8-4912-982222
126112	МАРКОВА В.П. имеет телефон 8-4912-683301
136169	ДЕНИСОВА Е.К. имеет телефон 8-4912-680305
080613	ЛУКАШИНА Р.М. имеет телефон 8-4912-254417
080047	ШУБИНА Т.П. имеет телефон 8-4912-257842
080270	ТИМОШКИНА Н.Г. имеет телефон 8-4912-321002

**Рис. 3.5.** Результат выполнения запроса с вычисляемыми столбцами

Существует возможность управлять количеством строк, возвращаемых запросом. Это делается в запросе SELECT с использованием следующей конструкции:

[FIRST m] [SKIP n] ,

где m и n - целочисленные выражения.

Здесь после зарезервированного слова FIRST указывается, что только первые m строк из возвращаемого набора данных нужно вывести, а оставшиеся – отбросить. Зарезервированное слово SKIP определяет параметр n – количество первых строк, которое нужно пропустить с начала набора данных, сформированного запросом.

Пусть, например, требуется вывести четыре строки из справочника абонентов, начиная со второй. Для этого можно использовать следующий запрос:

**SELECT FIRST 4 SKIP 1 \* FROM Abonent;**

Результат выполнения запроса представлен на рис. 3.6.

<b>ACCOUNTCD</b>	<b>STREETCD</b>	<b>HOUSENO</b>	<b>FLATNO</b>	<b>FIO</b>	<b>PHONE</b>
115705	3	1	82	МИЩЕНКО Е.В.	769975
015527	3	1	65	КОНЮХОВ В.С.	761699
443690	7	5	1	ТУЛУПОВА М.И.	214833
136159	7	39	1	СВИРИНА З.А.	350003

**Рис. 3.6.** Результат выполнения запроса на вывод определенных строк

Следует отметить, что конструкция `SELECT FIRST 0...` возвращает пустой результат.

Аналогом конструкции `FIRST...SKIP` является конструкция `ROWS...TO`, которая используется для ограничения количества строк, возвращаемых запросом, и имеет следующий формат:

`[ROWS k [TO r]].`

Целочисленные значения `k` и `r` показывают, что строки, начиная с `k`-й и заканчивая `r`-й, будут видны в ТРЗ.

Следует отметить, что конструкция `ROWS...TO` была введена в более позднем SQL-стандарте и поэтому обладает рядом преимуществ по сравнению с конструкцией `FIRST...SKIP` [21]. `ROWS...TO` может использоваться при объединении результатов нескольких запросов, в любых видах подзапроса, а также в запросах `UPDATE` и `DELETE` (будут рассмотрены далее).

Например, следующий запрос выведет из таблицы `Street` содержимое строк с третьей по шестую:

```
SELECT StreetCD, StreetNM FROM Street ROWS 3 TO 6;
```

Результат выполнения запроса представлен на рис. 3.7.

STREETCD	STREETNM
6	МОСКОВСКАЯ УЛИЦА
8	МОСКОВСКОЕ ШОССЕ УЛИЦА
4	ТАТАРСКАЯ УЛИЦА
5	ГАГАРИНА УЛИЦА

**Рис. 3.7.** Результат выполнения запроса на вывод строк с 3 по 6

Конструкция `ROWS...TO` всегда может быть заменена эквивалентной конструкцией `FIRST...SKIP`. Когда второе значение в конструкции `ROWS...TO` отсутствует, тогда конструкции `ROWS r` эквивалентна конструкция `FIRST r`. Когда и первое, и второе значения используются, тогда `ROWS k TO r` эквивалентна конструкция `FIRST (r - k + 1) SKIP (k - 1)`.

Если вывести четыре строки из справочника абонентов, начиная со второй, используя следующий запрос на основе конструкции `ROWS...TO`:

```
SELECT * FROM Abonent ROWS 2 TO 5;
```

то получится результат, совпадающий с результатом, приведенным на рис. 3.6.

Конструкцию `SKIP n` нельзя заменить конструкцией `ROWS...TO`.

Следует помнить, что хотя разрешено использование и `FIRST...SKIP`, и `ROWS...TO`, при попытке совместного их использования в одной команде будет выдано сообщение о синтаксической ошибке.

### 3.2.2. Предложение **WHERE**

Предложение `WHERE` используется для наложения условий на данные, выбираемые запросом `SELECT`. Предложение `WHERE` состоит из ключевого

слова WHERE, за которым следует условие поиска, определяющее, какие именно строки требуется вывести.

Предложение WHERE включает следующий набор условий для отбора строк:  
WHERE

[NOT] <условие\_поиска1> [[AND|OR][NOT] <условие\_поиска2>]...

Условие поиска применительно к однотабличным запросам определяется следующим образом [19]:

<условие\_поиска> ::=  
{ <значение> <операция\_сравнения> <значение1>  
| <значение> [NOT] BETWEEN <значение1> AND <значение2>  
| <значение> [NOT] LIKE 'шаблон' [ESCAPE 'символ\_пропуска']  
| <значение> [NOT] CONTAINING <значение1>  
| <значение> [NOT] STARTING WITH <значение1>  
| <значение> [NOT] IN ( <значение1> [ , <значение2> ... ] )  
| <значение> IS [NOT] NULL  
| <значение> IS [NOT] DISTINCT FROM <значение1> ,  
где  
<значение> ::= { столбец | константа | <выражение> | функция }.

### 3.2.2.1. Простое сравнение

Наиболее распространенным условием поиска в языке SQL является сравнение, которое реализуется следующей конструкцией:

<значение> <операция\_сравнения> <значение1>.

При сравнении происходит вычисление и сравнение двух значений для каждой строки данных. Значения могут быть простыми, например содержать одно имя столбца или константу, и сложными - арифметическими выражениями.

Например, для вывода номеров лицевых счетов абонентов и дат подачи ими непогашенных ремонтных заявок, можно использовать следующий запрос:

```
SELECT AccountCD, IncomingDate  
FROM Request WHERE Executed = 0;
```

Результат выполнения запроса представлен на рис. 3.8.

ACCOUNTCD	INCOMINGDATE
015527	28.02.1998
080270	31.12.2001
136159	01.04.2001
115705	28.12.2001

**Рис. 3.8.** Результат выполнения запроса с простым сравнением

При сравнении двух значений могут получиться три результата:

– если сравнение истинно, то результат проверки имеет значение TRUE;

- если сравнение ложно, то результат проверки имеет значение FALSE;
- если хотя бы одно из двух значений имеет значение NULL, то результатом проверки будет NULL.

При определении условий поиска необходимо помнить об обработке значений NULL. В трехзначной логике, принятой в SQL, условие поиска может иметь значение TRUE, FALSE или NULL. А в результаты запроса попадают только те строки, для которых условие поиска имеет значение TRUE.

### 3.2.2.2. Проверка на принадлежность диапазону значений

Другой формой условия поиска является проверка на принадлежность диапазону значений, которая реализуется с помощью ключевого слова BETWEEN. Синтаксис использования этого условия поиска следующий:

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

При этом проверяется, находится ли значение данных между двумя определенными значениями. В условие поиска входят три выражения. Первое выражение (слева от ключевого слова BETWEEN) определяет проверяемое значение; второе (после ключевого слова BETWEEN) и третье (после ключевого слова AND) выражения определяют нижний и верхний пределы проверяемого диапазона соответственно. При этом типы данных трех выражений должны быть сравнимы. Например, если необходимо найти номера лицевых счетов абонентов, у которых значения начислений за оказанные услуги лежат в диапазоне от 60 до 250, то соответствующий запрос будет выглядеть следующим образом:

```
SELECT AccountCD, Nachislsum
FROM NachislSumma
WHERE NachislSum BETWEEN 60 AND 250;.
```

Результат выполнения запроса представлен на рис. 3.9.

ACCOUNTCD	NACHISLSUM
115705	250,00
080047	80,00
080047	80,00
115705	250,00
443069	80,00
005488	62,13
080270	60,10

**Рис. 3.9.** Результат выполнения запроса с проверкой диапазона значений

При проверке на принадлежность диапазону нижний и верхний пределы считаются частью диапазона, поэтому в результаты запроса вошли лицевые счета, для которых значение начислений за оказанные услуги равно 250. Инвертированная проверка на принадлежность диапазону позволяет выбрать значения, которые лежат за пределами диапазона, например в следующем виде:



```
SELECT AccountCD FROM NachisSumma
WHERE NachisSum NOT BETWEEN 60 AND 250;.
```

### 3.2.2.3. Проверка на соответствие шаблону

Проверка на соответствие шаблону, которая осуществляется с помощью ключевого слова LIKE, позволяет определить, соответствует ли значение данных некоторому шаблону. Синтаксис использования этого условия поиска следующий:

<значение> [NOT] LIKE 'шаблон' [ESCAPE 'символ\_пропуска'].

Шаблон представляет собой строку, в которую могут входить один или более подстановочных знаков. Подстановочный знак процента (%) совпадает с любой последовательностью из нуля или более символов. Подстановочный знак подчеркивания (\_) совпадает с любым отдельным символом. При этом следует помнить, что пробел рассматривается как обычный символ. В операционной системе MS DOS знаку процента соответствует символ звездочки (\*), а знаку подчеркивания - знак вопроса (?). Подстановочные знаки можно помещать в любое место строки шаблона, и в одной строке может содержаться несколько подстановочных знаков.

При указании шаблона следует учитывать регистр символов. Так, например, LIKE '%А%' и LIKE '%а%' задают разные условия поиска.

Например, пусть необходимо выбрать из таблицы Abonent абонентов, фамилии которых начинаются с буквы С. Для условия поиска используется шаблон 'С%' следующим образом:

```
SELECT Fio FROM Abonent WHERE Fio LIKE 'С%';.
```

Результат выполнения запроса представлен на рис. 3.10.

FIO
СВИРИНА З.А.
СТАРОДУБЦЕВ Е.В.

**Рис. 3.10.** Результат выполнения запроса к таблице Abonent

Например, если точно не известна фамилия исполнителя ремонтных заявок ШЛЮКОВА М.К. (ШЛЮКОВ М.К. или ШЛАКОВ М.К.), можно воспользоваться шаблоном 'ШЛ\_КОВ М.К.%', чтобы получить информацию об интересующем исполнителе с помощью следующего запроса:

```
SELECT ExecutorCD, Fio FROM Executor
WHERE Fio LIKE 'ШЛ_КОВ М.К.%';.
```

Результат выполнения запроса представлен на рис. 3.11.

EXECUTORCD	FIO
4	ШЛЮКОВ М.К.

**Рис. 3.11.** Результат выполнения запроса к таблице Executor

В учебной БД для описания поля Fio используется тип VARCHAR(20), из чего следует, что пробелы в конце строки отрезаются автоматически, и поэтому в предыдущем примере можно было воспользоваться шаблоном 'ШЛ\_КОВ М.К.' (без знака процента в конце). Однако в случае, если поле БД имеет тип CHAR(n), использование знака процента в конце строки шаблона необходимо для того, чтобы строки с такими полями (дополненными справа пробелами до общего количества символов n) были включены в результат выполнения запроса.

При проверке строк на соответствие шаблону может оказаться, что подстановочные знаки входят в строку символов в качестве литералов. Например, нельзя проверить, содержится ли знак процента в строке, включив его в шаблон, поскольку он будет считаться подстановочным знаком. В стандарте ANSI/ISO определен способ проверки наличия в строке литералов, использующихся в качестве подстановочных знаков. Для этого применяются символы пропуска. Когда в шаблоне встречается такой символ, то символ, следующий непосредственно за ним, считается не подстановочным знаком, а литералом (т.е. происходит пропуск символа). Непосредственно за символом пропуска может следовать либо один из двух подстановочных знаков, либо сам символ пропуска, поскольку он тоже приобретает в шаблоне особое значение. Символ пропуска определяется в виде строки, состоящей из одного символа, и фразы ESCAPE. Например, чтобы найти фамилии абонентов, начинающихся со знака процента, нужно выполнить следующий запрос:

```
SELECT Fio FROM Abonent  
WHERE Fio LIKE '$%%' ESCAPE '$';
```

Этот запрос не имеет результатов, т.к. в таблице Abonent нет абонентов, фамилии которых начинаются с символа процента. Первый символ процента в шаблоне, следующий за символом пропуска \$, считается литералом, второй считается подстановочным знаком.

#### **3.2.2.4. Проверка на наличие последовательности символов**

Предикат CONTAINING проверяет, содержит ли строковая величина, указанная слева от него, последовательность символов, указанную справа. Синтаксис использования этого условия поиска следующий:

```
<значение> [NOT] CONTAINING <значение1>
```

где

<значение> - любое строковое выражение;

<значение1> - строковая константа.

Поиск CONTAINING является чувствительным к регистру. Предикат CONTAINING может быть использован для алфавитно-цифрового поиска в числах и датах. Следующий пример выводит информацию обо всех неисправностях, в названиях которых встречается 'Неисправ':

```
SELECT * FROM Disrepair  
WHERE FailureNM CONTAINING 'Неисправ';
```

Результат выполнения запроса представлен на рис. 3.12.

FAILURECD	FAILURENM
4	Неисправна печная горелка
5	Неисправен газовый счетчик

**Рис. 3.12.** Результат работы предиката CONTAINING

NOT CONTAINING используется для отбора строк, в которых заданное значение не включает указанную строковую константу.

### 3.2.2.5. Проверка на совпадение с началом строки

Предикат STARTING WITH проверяет, совпадают ли начальные символы строкового выражения, стоящего слева от него, с оговоренной строкой символов, указанной справа. Синтаксис использования этого условия поиска следующий:

<значение> [NOT] STARTING WITH <значение1>,

где <значение> - любое строковое выражение, а <значение1> - строковая константа.

Следующий запрос выводит данные обо всех абонентах, фамилия которых начинается с буквы Т:

```
SELECT * FROM Abonent WHERE Fio STARTING WITH 'Т';
```

Результат выполнения запроса представлен на рис. 3.13.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
443690	7	5	1	ТУЛУПОВА М.И.	214833
080270	6	35	6	ТИМОШКИНА Н.Г.	321002

**Рис. 3.13.** Результат работы предиката STARTING WITH

Следует отметить, что предикат STARTING WITH является чувствительным к регистру.

### 3.2.2.6. Проверка на членство в множестве

Еще одним распространенным условием поиска является проверка на членство в множестве, которое реализуется с помощью ключевого слова IN. Синтаксис использования этого условия поиска следующий:

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

В этом случае проверяется, соответствует ли значение данных какому-либо значению из заданного списка. Например, чтобы вывести неисправности с кодами, равными 1, 5 и 12, можно воспользоваться условием поиска с проверкой на членство в множестве (1, 5, 12). Таким образом, соответствующий запрос к таблице Disrepair будет выглядеть следующим образом:

```
SELECT * FROM Disrepair WHERE FailureCD IN (1,5,12);
```

Результат выполнения запроса представлен на рис. 3.14.

FAILURECD	FAILURENM
1	Засорилась водогрейная колонка
5	Неисправен газовый счетчик
12	Неизвестна

**Рис. 3.14.** Результат выполнения запроса с проверкой вхождения в множество

С помощью конструкции NOT IN можно убедиться в том, что значение данных не является членом заданного множества. Если результатом проверяемого выражения является NULL, то проверка IN также возвращает NULL. Все значения в списке заданных значений должны иметь один и тот же тип данных, который должен быть сравним с типом данных проверяемого выражения.

### 3.2.2.7. Проверка значения на NULL

Признак NULL обеспечивает возможность применения трехзначной логики в условиях поиска. Результатом применения любого условия поиска может быть TRUE, FALSE или NULL (в случае, когда в одном из столбцов содержится NULL). Иногда бывает необходимо явно проверять значения столбцов на NULL и непосредственно обрабатывать их. Для этого в языке SQL имеется специальная проверка IS NULL. Синтаксис использования этого условия поиска следующий:

<значение> IS [NOT] NULL .

Например, необходимо вывести номера лицевых счетов абонентов и даты подачи ими заявок, по которым не выполнены ремонтные работы, т.е. поле ExecutionDate содержит NULL. Для этого можно использовать следующий запрос:

```
SELECT AccountCD, IncomingDate  
FROM Request  
WHERE ExecutionDate IS NULL;
```

Результат выполнения запроса представлен на рис. 3.15.

ACCOUNTCD	INCOMINGDATE
080270	31.12.2001
115705	28.12.2001

**Рис. 3.15.** Результат выполнения запроса с проверкой значения на NULL

В отличие от условий поиска, описанных выше, проверка на NULL не может вернуть NULL в качестве результата. Она всегда возвращает TRUE или

FALSE. Следует отметить, что *нельзя проверить значение на равенство NULL с помощью простой операции сравнения*, например:

```
SELECT AccountCD, IncomingDate
FROM Request WHERE ExecutionDate = NULL;
```

Запрос не будет выдавать синтаксическую ошибку (как это было бы в более ранних версиях Firebird), так как теоретически литерал NULL может участвовать во всех выражениях ( $A = NULL$ ,  $B > NULL$ ,  $A + NULL$ ,  $B \parallel NULL$  и т.д.). Однако получится неправильный результат (NULL), так как если в операции сравнения одно из выражений имеет значение NULL, то и результат будет NULL.

### 3.2.2.8. Проверка двух значений на отличие

Для проверки значений на отличие используется следующий синтаксис:

```
<значение> IS [NOT] DISTINCT FROM <значение1>.
```

Предикат DISTINCT аналогичен предикату равенства с тем лишь различием, что считает два признака NULL не различающимися (возвращает TRUE при  $NULL=NULL$ ). Поскольку предикат DISTINCT считает, что два признака NULL не различаются, то он никогда не возвращает неизвестное значение. Подобно предикату IS [NOT] NULL предикат DISTINCT в качестве результата возвращает только TRUE или FALSE.

Например, требуется вывести всю информацию о ремонтных заявках с кодом неисправности, равным 1, даты выполнения которых отличаются от 20.12.2001. Соответствующий запрос будет выглядеть следующим образом:

```
SELECT * FROM Request
WHERE FailureCD=1 AND
      ExecutionDate IS DISTINCT FROM '20.12.2001';
```

Результат выполнения запроса представлен на рис. 3.16, из которого следует, что признак NULL включен в TP3.

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
2	115705	3	1	07.08.2001	12.08.2001	1
5	080270	4	1	31.12.2001	<null>	0
9	136169	2	1	06.11.2001	08.11.2001	1

**Рис. 3.16.** Результат выполнения запроса на проверку отличия

### 3.2.2.9. Составные условия поиска

Рассмотренные в предыдущих пунктах условия поиска являются простыми. С помощью правил логики эти простые условия можно объединять в более сложные.

СУБД Firebird предоставляет три вида логических операций [18]:

- NOT задает отрицание условия поиска, к которому применяется, и имеет наивысший приоритет. Используется следующий синтаксис:

NOT <условие\_поиска>;

- AND создает сложный предикат, объединяя два или более условий поиска, каждое из которых должно быть истинным, чтобы был истинным и весь предикат. Данная операция является следующей по приоритету после NOT. Используется следующий синтаксис:

<условие\_поиска1> AND <условие\_поиска2> ...;

- OR создает сложный предикат, объединяя два или более условий поиска, из которых хотя бы одно должно быть истинным, чтобы был истинным и весь предикат. Является последней по приоритету из трех логических операций и имеет следующий синтаксис:

<условие\_поиска1> OR <условие\_поиска2> ....

Вычисления логических значений в составных условиях поиска задаются таблицей истинности (см. табл. 3.1).

**Таблица 3.1.** Логические результаты условий поиска AND и OR

Y1 (условие_поиска1)	Y2 (условие_поиска2)	Y1 AND Y2	Y1 OR Y2
TRUE	TRUE	TRUE	TRUE
TRUE (FALSE)	FALSE (TRUE)	FALSE	TRUE
TRUE (NULL)	NULL (TRUE)	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE (NULL)	NULL (FALSE)	FALSE	NULL
NULL	NULL	NULL	NULL

Следует обратить внимание на то, что условия поиска, объединенные с помощью ключевых слов AND, OR и NOT, сами могут быть составными.

Допустим, что необходимо найти фамилии всех абонентов, которые проживают на улицах с кодами от 3 до 6 или фамилии которых содержат букву 'Л'. Для вывода требуемой информации нужно выполнить следующий запрос:

```
SELECT Fio FROM Abonent  
WHERE (Streetcd BETWEEN 3 AND 6) OR (Fio LIKE '%Л%');
```

Результат выполнения запроса представлен на рис. 3.17.

<b>FIO</b>
АКСЕНОВ С.А.
МИЩЕНКО Е.В.
КОНЮХОВ В.С.
ТУЛУПОВА М.И.
СТАРОДУБЦЕВ Е.В.
ШМАКОВ С.В.
МАРКОВА В.П.
ДЕНИСОВА Е.К.
ЛУКАШИНА Р.М.
ТИМОШКИНА Н.Г.

**Рис. 3.17.** Результат выполнения запроса к таблице Abonent

Например, необходимо извлечь все данные об оплатах, которые были произведены после 13 июня 2001 года и значения оплат которых превышают 60. Одновременно с этим вывести все данные об оплатах, которые были сделаны абонентом с лицевым счётом "005488" до 2000 года. Для решения данной задачи можно использовать следующий запрос:

```
SELECT * FROM PaySumma
WHERE (Paydate > '13.06.2001' AND PaySum>60) OR
(Paydate < '01.01.2000' AND AccountCD = '005488');
```

Результат выполнения запроса представлен на рис. 3.18.

PAYFACTCD	ACCOUNTCD	GAZSERVICECD	PAYSUM	PAYDATE	PAYMONTH	PAYYEAR
3	005488	2	56,000	06.05.1999	4	1999
5	115705	2	250,000	03.10.2001	9	2001
10	080047	2	80,000	21.11.2001	10	2001
16	443069	2	80,000	03.10.2001	9	2001

**Рис. 3.18.** Результат выполнения запроса к таблице PaySumma

### 3.2.3. Функции в SQL

#### 3.2.3.1. Классификация функций

Функции SQL подобны любым другим запросам языка в том смысле, что они производят действия с данными и возвращают результат в качестве своего значения. Имеется два основных класса функций в СУБД Firebird: встроенные и определяемые пользователем.

**Встроенными** являются функции, предопределенные в языке SQL СУБД Firebird. В SQL определено множество встроенных функций различных категорий [16, 20, 21]. Эти функции делятся на три основные группы:

- скалярные функции;
- агрегатные функции;
- функции для списка значений.

*Скалярные функции* (их еще называют однострочными) обрабатывают одиночное значение и возвращают также одно значение. Скалярные функции разрешается использовать везде, где допускается применение выражений.

Скалярные функции бывают следующих категорий:

- строковые функции, которые выполняют определенные действия над строками и возвращают строковые или числовые значения;
- числовые функции, которые возвращают числовые значения на основании заданных в аргументе значений того же типа;
- функции времени и даты, которые выполняют различные действия над входными значениями времени и даты и возвращают строковое, числовое значение или значение в формате даты и времени;
- функция преобразования типа.

Особое место среди встроенных скалярных функций языка SQL занимают *функции вывода*, которые являются разновидностью CASE-выражений. В качестве функций вывода используются функции COALESCE, NULLIF, IF и DECODE. Возвращаемый этими функциями результат меняется в зависимости от значения, которое обрабатывается функцией. Функции вывода будут подробно рассмотрены при изучении дополнительных возможностей выбора вариантов вывода в предложении SELECT.

*Агрегатные функции* используются для получения обобщающих значений. Они, в отличие от скалярных функций, оперируют значениями столбцов множества строк. К агрегатным функциям относятся такие функции, как SUM, вычисляющая итог, MAX и MIN, возвращающие наибольшее и наименьшее значения соответственно, AVG, вычисляющая среднее значение, и COUNT, вычисляющая количество значений в столбце.

*Функции для списка значений* представлены тремя функциями – MAXVALUE, MINVALUE и LIST. Функции MAXVALUE и MINVALUE в чем-то похожи на агрегатные функции MAX и MIN, однако выбирают максимальное и минимальное значение не из множества значений строк в одном столбце, а из значений, заданных в списке аргументов. Функция LIST объединяет обрабатываемые значения в единую строку.

**Функции, определяемые пользователем (UDF)**, являются вспомогательными программами, написанными на языке программирования, таком как C, C++ или Pascal, и скомпилированными как совместно используемые двоичные библиотеки – DLL [18]. Внешние функции можно использовать в выражениях, так же как и встроенные функции SQL. Как и встроенные функции, они могут возвращать значения для переменных или выражений SQL в хранимых процедурах и триггерах.

СУБД Firebird предоставляет две готовые к использованию библиотеки UDF: `ib_udf` и `fbudf`. Firebird загружает UDF из библиотек, находящихся в каталоге `udf` каталога инсталляции или в других каталогах, указанных в параметре `UdfAccess` в файле конфигурации Firebird. Когда пользовательская функция написана, скомпилирована и инсталлирована в соответствующий каталог на сервере, она должна быть объявлена для базы данных, чтобы ее можно было использовать как функцию SQL. Для объявления внешней функции используется оператор `DECLARE EXTERNAL FUNCTION`. Можно объявить функцию также с использованием любого интерактивного инструмента SQL или скрипта. После того как функция будет объявлена для любой базы данных на сервере, содержащая ее библиотека будет загружаться при первом же обращении приложения к любой функции, включенной в эту библиотеку. Необходимо объявить каждую функцию, которую нужно использовать, для каждой базы данных, которая будет использоваться.

Рассмотрим подробнее встроенные функции СУБД Firebird 2.1.



### 3.2.3.2. Скалярные функции

#### 3.2.3.2.1. Строковые функции

Эти функции используют в качестве аргумента строку символов, а в качестве результата возвращают также символьную строку или числовое значение.

Для выделения подстроки указанной длины из строкового выражения, начиная с заданной позиции, используется функция **SUBSTRING**, имеющая следующий формат:

**SUBSTRING** (<строковое\_выражение> FROM позиция [FOR длина]),

где позиция – позиция, начиная с которой выполняется выделение, например 1 для первого символа в строке;

длина – количество выделяемых символов.

Например, для вывода номеров лицевого счета абонентов и первых трех символов их фамилии можно использовать следующий запрос:

```
SELECT A.AccountCD, SUBSTRING (A.Fio FROM 1 for 3) AS Fio3  
FROM Abonent A;
```

Результат выполнения запроса представлен на рис. 3.19.

<b>ACCOUNTCD</b>	<b>FIO3</b>
005488	АКС
115705	МИЩ
015527	КОН
443690	ГУЛ
136159	СВИ
443069	СТА
136160	ШМА
126112	МАР
136169	ДЕН
080613	ЛУК
080047	ШУБ
080270	ТИМ

**Рис. 3.19.** Результат работы функции SUBSTRING

Следует отметить, что в качестве аргументов позиция и длина в **SUBSTRING** можно использовать и выражения (в том числе со скалярными функциями, возвращающими числовой результат). Также в качестве аргументов **SUBSTRING** могут использоваться подзапросы, возвращающие единственное значение (<скалярный\_подзапрос>).

Следует учесть, что длина результата будет такой же, как и длина первого аргумента (<строковое выражение>, из которого выделяются символы). Длина поля Fio таблицы Abonent равна 20 [VARCHAR(20)], длина нового поля будет также иметь длину 20, а не 3 символа.

Для выделения строки в обратной последовательности (начиная с конца) используется функция **REVERSE**. Функция имеет следующий формат:

**REVERSE** (<строковое\_выражение>).

Например, для вывода информации об абонентах, имеющих инициалы Е.В., можно использовать следующий запрос:

```
SELECT * FROM Abonent
WHERE REVERSE (Fio) STARTING WITH REVERSE ('E.V.);
```

Результат выполнения запроса представлен на рис. 3.20.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
115705	3	1	82	МИЩЕНКО Е.В.	769975
443069	4	51	55	СТАРДУБЦЕВ Е.В.	683014

**Рис. 3.20.** Результат работы функции REVERSE

Функции **LEFT** и **RIGHT** используются для выделения нужного количества символов из начала или конца определенной строки соответственно и имеют следующий формат:

**LEFT** (<строковое\_выражение>, длина),

**RIGHT** (<строковое\_выражение>, длина),

где <строковое\_выражение> – выражение, из которого будут выделяться символы;

длина – количество выделяемых символов в начале (для **LEFT**) или конце (для **RIGHT**) строки.

Например, предыдущий запрос, реализованный с помощью функции **REVERSE**, можно реализовать с помощью функции **RIGHT** следующим образом:

```
SELECT * FROM Abonent WHERE RIGHT (Fio, 4) = 'E.V.';
```

Результат выполнения запроса будет совпадать с результатом, представленным на рис. 3.20.

Например, для вывода Fio абонентов, проживающих на улице с кодом 3, и первых 4-х цифр из номеров их лицевых счетов можно использовать следующий запрос:

```
SELECT Fio, LEFT (AccountCD, 4) FROM Abonent
WHERE StreetCD = 3;
```

Результат выполнения запроса представлен на рис. 3.21.

FIO	RIGHT
АКСЕНОВ С.А.	0054
МИЩЕНКО Е.В.	1157
КОНЮХОВ В.С.	0155

**Рис. 3.21.** Результат работы функции RIGHT

Существует ряд функций для замены части исходной строки на другую последовательность символов.

Функция **OVERLAY** заменяет в исходной строке подстроку, начинающуюся с номера позиция и имеющую размер длина, на значение строки для замены. Функция имеет следующий формат:

```
OVERLAY ( <исходная_строка> PLACING <строка_для_замены>
FROM позиция [ FOR длина ] ),
```

где <исходная\_строка> и <строка\_для\_замены> могут представлять собой строковое выражение.

Если длина не указана, то по умолчанию принимается длина в символах строки для замены (CHAR\_LENGTH (<строка\_для\_замены>)).

Функция **OVERLAY** эквивалентна следующему выражению с использованием функции **SUBSTRING**:

```
SUBSTRING (<исходная_строка> FROM 1 FOR (позиция - 1))
|| <строка_для_замены>
|| SUBSTRING (<исходная_строка> FROM (позиция + длина)).
```

Функция **REPLACE** заменяет все вхождения <подстроки> в <строковое\_выражение> на указанную <строку\_для\_замены>. Имеет следующий формат:

```
REPLACE (<строковое_выражение>, <подстрока>, <строка_для_замены>).
```

Например, для замены в названии неисправностей слова "плиты" на словосочетание "газовой плиты" можно выполнить следующий запрос:

```
SELECT REPLACE (FailureNM, 'плиты', 'газовой плиты')
FROM Disrepair;
```

Результат выполнения запроса представлен на рис. 3.22.

<b>REPLACE</b>
Засорилась водогрейная колонка
Не горит АГВ
Течет из водогрейной колонки
Неисправна печная горелка
Неисправен газовый счетчик
Плохое поступление газа на горелку газовой плиты
Туго поворачивается пробка крана газовой плиты
При закрытии краника горелка газовой плиты не гаснет
Неизвестна

**Рис. 3.22.** Результат работы функции **REPLACE**

К скалярным функциям относится также функция **TRIM**. Эта функция возвращает строку аргумента, удаляя символы (по умолчанию - пробелы) из начала и/или конца строки.

Для вызова функции используется следующий синтаксис:

```
TRIM ( [ [LEADING | TRAILING | BOTH] [ <удаляемая_подстрока> ]
FROM ] <строковое_выражение> ).
```

LEADING указывает на то, что надо удалить указанную подстроку из начала строки, TRAILING – из конца строки, BOTH - из начала и конца строки.

При использовании функции следует учитывать следующее:

- 1) если LEADING, TRAILING или BOTH не указаны, то принимается BOTH по умолчанию;
- 2) если <удаляемая\_подстрока> не определена, то за нее принимается по умолчанию пустая строка;
- 3) если LEADING, или TRAILING, или BOTH и/или <удаляемая\_подстрока> указаны, то после слова FROM должно быть обязательно указано <строковое\_выражение>, из которого удаляются символы;
- 4) указание строки, из которой удаляются символы, после слова FROM не может быть использовано самостоятельно (если LEADING, TRAILING, BOTH и <удаляемая\_подстрока> не указаны).

Например, для вывода данных из таблицы Street, указывая название улицы без слова "УЛИЦА", можно использовать следующий запрос:

```
SELECT StreetCD,  
       TRIM (BOTH 'УЛИЦА' FROM StreetNM) AS Str_Name  
FROM Street;
```

Результат выполнения запроса представлен на рис. 3.23.

STREETCD	STR_NAME
3	ВОЙКОВ ПЕРЕУЛОК
7	КУТУЗОВА
6	МОСКОВСКАЯ
8	МОСКОВСКОЕ ШОССЕ
4	ТАТАРСКАЯ
5	ГАГАРИНА
1	ЦИОЛКОВСКОГО
2	НОВАЯ

Рис. 3.23. Результат работы функции TRIM

При указании удаляемой подстроки следует учитывать регистр символов (строчные или прописные буквы). Если в предыдущем примере вместо 'УЛИЦА' ввести 'улица', то запрос выдаст неверный результат, так как все названия улиц в таблице Street записаны в верхнем регистре символов.

Существуют функции **LPAD** и **RPAD**, которые дополняют строку аргумента слева (LPAD) или справа (RPAD) указанной последовательностью символов (<строка\_заполнитель>) до заданного размера (длина). В случае если <строка\_заполнитель> не указана, для дополнения используется пробел. Строка-заполнитель обрезаются, когда результирующая строка достигает заданной длины. Функции имеют следующий формат:

```
LPAD( <строковое_выражение>, длина [, <строка_заполнитель> ] ),  
RPAD( <строковое_выражение>, длина [, <строка_заполнитель> ] ).
```

Например, требуется вывести номера лицевых счетов абонентов и Fio, дополненные справа знаком звездочки (\*) до длины 20 символов. Запрос будет выглядеть следующим образом:

```
SELECT AccountCD, RPAD (Fio, 20, '*') FROM Abonent;
```

Результат выполнения запроса представлен на рис. 3.24.

ACCOUNTCD	RPAD
005488	АКСЕНОВ С.А.*****
136169	ДЕНИСОВА Е.К.*****
015527	КОНЮХОВ В.С.*****
080613	ЛУКАШИНА Р.М.*****
126112	МАРКОВА В.П.*****
115705	МИЩЕНКО Е.В.*****
136159	СВИРИНА З.А.*****
443069	СТАРОДУБЦЕВ Е.В.****
080270	ТИМОШКИНА Н.Г.*****
443690	ТУЛУПОВА М.И.*****
136160	ШМАКОВ С.В.*****
080047	ШУБИНА Т.П.*****

**Рис. 3.24.** Результат работы функции RPAD

Существуют функции, которые выполняют преобразования между регистрами символов. Функция **UPPER** преобразует все символы строки в верхний регистр. Имеет следующий синтаксис:

```
UPPER (<значение>),
```

где <значение> - преобразуемый столбец, переменная или выражение строкового типа.

Если набор символов и последовательность сортировки поддерживают преобразование в верхний регистр (например, WIN1251), то функция возвращает строку, в которой все символы преобразованы в верхний регистр. Строка имеет ту же длину, что и входное <значение>. Для наборов символов, не поддерживающих преобразование в верхний регистр (это, в частности, кодировка по умолчанию NONE), функция возвращает неизменное входное значение. Следующий запрос выводит названия услуг газоснабжения заглавными буквами:

```
SELECT UPPER(GazserviceNM) FROM Services;
```

Результат выполнения запроса представлен на рис. 3.25.

UPPER
ДОСТАВКА ГАЗА
ЗАЯВОЧНЫЙ РЕМОНТ ГО

**Рис. 3.25.** Результат работы функции UPPER

Противоположной функцией является **LOWER**, которая преобразует все символы строки в нижний регистр. Имеет следующий синтаксис:

LOWER (<значение>),

где <значение> - преобразуемый столбец, переменная или выражение строкового типа.

Для определения символа по известному коду используется функция **ASCII\_CHAR**. Функция имеет следующий формат:

ASCII\_CHAR (код\_символа) .

Код должен лежать в диапазоне от 0 до 255. Следует учесть, что функция возвращает символ для кодировки NONE.

Рассмотренные выше строковые функции возвращают результат в виде строки символов. Существует также ряд строковых функций, которые в качестве результата возвращают числовое значение.

Для определения кода первого символа в указанной строке используется функция **ASCII\_VAL**. Функция имеет следующий формат:

ASCII\_VAL (<строка>) .

Функция вернет ноль, если указана пустая строка.

Для определения позиции первого вхождения заданной подстроки в строку можно использовать функцию **POSITION**. Функция имеет следующий формат:

POSITION (<подстрока> IN <строковое\_выражение>) .

Функция возвращает ноль, если подстрока отсутствует внутри строки.

Например, для вывода номеров лицевого счета и фамилий тех абонентов, у которых в их фамилиях вторая буква У, можно использовать следующий запрос:

```
SELECT AccountCD, Fio FROM Abonent  
WHERE POSITION ('У' IN Fio) = 2;
```

Результат выполнения запроса представлен на рис. 3.26.

ACCOUNTCD	FIO
443690	ГУЛУПОВА М.И.
080613	ЛУКАШИНА Р.М.
080047	ШУБИНА Т.П.

Рис. 3.26. Результат работы функции POSITION

Для определения размера строки применяются функции **BIT\_LENGTH**, **CHAR[ACTER]\_LENGTH**, **OCTET\_LENGTH** .

Функция BIT\_LENGTH возвращает длину строки в битах, функция CHAR[ACTER]\_LENGTH - в символах, а функция OCTET\_LENGTH - в байтах.

Эти три функции имеют одинаковый синтаксис:

{ BIT\_LENGTH | CHAR[ACTER]\_LENGTH | OCTET\_LENGTH }  
( <строковое\_выражение> ).

Например, вывести названия услуг газоснабжения и длины полей названия в символах и в битах можно с помощью следующего запроса:

```
SELECT GazServiceNM, CHAR_LENGTH (GazServiceNM),
      BIT_LENGTH (GazServiceNM)
FROM Services;
```

Результат выполнения запроса представлен на рис. 3.27.

GAZSERVICENM	CHAR_LENGTH	BIT_LENGTH
Доставка газа	13	104
Заявочный ремонт ГО	19	152

**Рис. 3.27.** Результаты работы функций CHAR\_LENGTH и BIT\_LENGTH

Проверить, что тип VARCHAR предусматривает автоматическое отбрасывание символов пробела можно помощью следующего запроса, использующего функции CHAR\_LENGTH и TRIM:

```
SELECT FIRST 3 FailureCD,
      CHAR_LENGTH (FailureNM),
      CHAR_LENGTH (TRIM(FailureNM))
FROM Disrepair;
```

Результат выполнения запроса представлен на рис. 3.28.

FAILURECD	CHAR_LENGTH	CHAR_LENGTH1
1	30	30
2	12	12
3	28	28

**Рис. 3.28.** Результат работы функции CHAR\_LENGTH

Во втором столбце выводятся длины значений поля FailureNM в символах, а в третьем – длины значений этого же поля, но с удаленными символами пробелов. Можно заметить, что значения во втором и третьем столбце совпадают, так как тип поля FailureNM – VARCHAR(50) и символы пробелов автоматически отбрасываются.

### 3.2.3.2.2. Числовые функции

Эти функции возвращают числовые значения на основании значений того же типа, заданных в аргументе. Числовые функции используются для обработки данных, а также в условиях поиска. Стандартные числовые функции СУБД Firebird 2.1 перечислены в табл. 3.2.

**Таблица 3.2. Числовые функции**

<b>Функция</b>	<b>Описание</b>
RAND()	Случайное число от 0 до 1
ABS (число)	Абсолютное значение
SIGN (число)	Знаковая функция (возвращает 1 для положительного числа, 0 – для нуля, -1 – для отрицательного числа)
MOD (делимое, делитель)	Остаток от деления
LOG (основание, число)	Логарифм числа по указанному основанию
LN (число)	Натуральный логарифм числа
LOG10 (число)	Десятичный логарифм числа
EXP (число)	Экспоненциальная функция (е в степени аргумента)
PI()	Константа $\pi = 3.1459\dots$
POWER (число, степень)	Возведение числа в степень
SQRT (число)	Квадратный корень
FLOOR (число)	Округление до целого числа вниз
CEIL   CEILING (число)	Округление до целого числа вверх
ROUND (число, точность)	Округление до указанного количества знаков после запятой
TRUNC (число)	Целая часть числа
HASH(<значение>)	Хэш-функция (рандомизация значения)
<i>Тригонометрические функции</i>	
SIN (число)	Синус (аргумент задается в радианах)
COS (число)	Косинус (угол определяется в радианах, результат в диапазоне от -1 до 1)
TAN (число)	Тангенс (аргумент задается в радианах)
COT (число)	Котангенс
ASIN (число)	Арксинус (число должно быть в диапазоне от -1 до 1, результат от $-\pi/2$ до $\pi/2$ )
ACOS (число)	Арккосинус (число должно быть в диапазоне от -1 до 1, результат от 0 до $\pi$ )
ATAN (число)	Арктангенс (возвращает результат в диапазоне от $-\pi/2$ до $\pi/2$ )
SINH (число)	Гиперболический синус
COSH (число)	Гиперболический косинус
TANH (число)	Гиперболический тангенс
ATAN2 (число1, число2)	Арктангенс в градусах, вычисляемый как арктангенс результата деления одного тангенса на другой – ATAN(число1/число2). Возвращает результат в диапазоне $(-\pi; \pi]$



**Таблица 3.2. Окончание**

<i>Логические функции</i>	
BIN_AND (число [,число...])	Логическое 'И' на всех аргументах
BIN_OR (число [,число>..])	Логическое 'ИЛИ' на всех аргументах
BIN_XOR (число [,число...])	Исключающее 'ИЛИ' на всех аргументах
BIN_SHL (число, число)	Двоичный сдвиг влево
BIN_SHR (число, число)	Двоичный сдвиг вправо

### 3.2.3.2.3. Функции даты и времени

Как уже отмечалось, эти функции выполняют различные действия над входными значениями времени и даты и возвращают строковое, числовое значение или значение в формате даты и времени.

Для выделения значений дня, месяца и года из даты используется функция **EXTRACT**.

Синтаксис этой функции следующий:

**EXTRACT( { DAY | MONTH | YEAR } FROM <значение> ),**

где <значение> - любое выражение, возвращающее результат типа «дата-время».

Пусть, например, требуется для каждой ремонтной заявки, год поступления которой отличается от 2001, указать ее номер, день, месяц и год даты поступления. Для этого можно использовать следующий запрос:

```
SELECT RequestCD,
       EXTRACT(DAY FROM IncomingDate)
       AS IncomingDay,
       EXTRACT (MONTH FROM IncomingDate)
       AS IncomingMonth,
       EXTRACT (YEAR FROM IncomingDate)
       AS IncomingYear
FROM Request
WHERE EXTRACT (YEAR FROM IncomingDate)
      IS DISTINCT FROM 2001;
```

Результат выполнения запроса представлен на рис. 3.29.

REQUESTCD	INCOMINGDAY	INCOMINGMONTH	INCOMINGYEAR
3	28	2	1998
7	20	10	1998
11	12	1	1999
13	4	9	2000
14	4	4	1999
15	20	9	2000
18	28	12	1999

**Рис. 3.29. Результат работы функции EXTRACT**

Для получения значений текущей даты и системного времени сервера используются функции **CURRENT\_TIMESTAMP**, **CURRENT\_DATE** и **CURRENT\_TIME**.

Функция **CURRENT\_TIMESTAMP** возвращает значение типа **TIMESTAMP** (дату и время вместе).

Например, вывести данные таблицы **Services** и текущие дату и время можно помощью следующего запроса:

```
SELECT S.*, CURRENT_TIMESTAMP FROM Services S;
```

Результат выполнения запроса представлен на рис. 3.30.

GAZSERVICECD	GAZSERVICENM	CURRENT_TIMESTAMP
1	Доставка газа	04.05.2007 11:23
2	Заявочный ремонт ГО	04.05.2007 11:23

**Рис. 3.30.** Результат работы функции **CURRENT\_TIMESTAMP**

Для возврата текущей даты сервера можно использовать функцию **CURRENT\_DATE**, а для возврата системного времени сервера – функцию **CURRENT\_TIME**.

Функция **DATEADD** возвращает значение типа дата, время или дата/время, увеличенное или уменьшенное (если количество отрицательное) по сравнению с исходным значением на заданное количество лет, месяцев, дней, недель, часов, минут или секунд. Функция имеет следующий формат:

```
DATEADD ( количество <временной_отрезок> FOR <значение> )
```

или

```
DATEADD (<временной_отрезок>, количество, <значение> ),
```

где количество - количество прибавляемых или вычитаемых единиц;

```
<временной_отрезок> ::= { YEAR | MONTH | DAY | WEEKDAY | HOUR | MINUTE | SECOND };
```

<значение> - значение типа дата, время или дата/время, которое увеличивается или уменьшается.

Следует отметить следующее:

- **YEAR**, **MONTH**, **DAY** и **WEEKDAY** в качестве временного отрезка не могут использоваться со значениями типа время (например, тип **TIME**);
- **HOUR**, **MINUTE** и **SECOND** в качестве временного отрезка не могут использоваться со значениями типа дата (например, тип **DATE**);
- все значения временного отрезка могут быть использованы для типов дата/время (тип **TIMESTAMP**).

Например, требуется вывести даты регистрации ремонтных заявок с кодом неисправности, равным 1, и даты через две недели после их регистрации. Запрос будет выглядеть следующим образом:

```
SELECT IncomingDate,  
       DATEADD (2 WEEKDAY FOR IncomingDate) AS Exec_Limit  
FROM Request  
WHERE FailureCD = 1;
```

Результат выполнения предыдущего запроса представлен на рис. 3.31.

<b>INCOMINGDATE</b>	<b>EXEC_LIMIT</b>
17.12.2001	31.12.2001
07.08.2001	21.08.2001
31.12.2001	14.01.2002
06.11.2001	20.11.2001

**Рис. 3.31.** Результат работы функции DATEADD

Для определения величины временного промежутка от первого заданного значения типа дата, время или дата/время до второго может использоваться функция **DATEDIFF**. Данная функция возвращает значение типа BIGINT и имеет следующий формат:

DATEDIFF ( <временной\_отрезок> FROM <значение1> FOR <значение2> )

или

DATEDIFF ( <временной\_отрезок>, <значение1>, <значение2> ),

где <временной\_отрезок> имеет тот же синтаксис, что и в функции DATEADD.

Следует отметить следующее:

- функция возвращает положительное число, если <значение2> превышает <значение1>, отрицательное - если <значение1> превышает <значение2>, и ноль - если значения равны;
- если результат вычисления дробный, то выводится округленное значение;
- сравнение значения типа DATE со значением типа TIME недопустимо;
- как и для функции DATEADD, определенные временные отрезки могут использоваться только с соответствующим им типом.

Например, требуется для заявок, поданных абонентом с лицевым счетом '115705', вывести количество недель, прошедших с даты регистрации заявки до момента ее выполнения. Для этого можно использовать следующий запрос:

```
SELECT RequestCD, DATEDIFF (WEEKDAY FROM IncomingDate
                             FOR ExecutionDate) AS Interval
FROM Request WHERE AccountCD = '115705';.
```

Результат выполнения запроса представлен на рис. 3.32.

<b>REQUESTCD</b>	<b>INTERVAL</b>
2	0
15	0
16	<null>
17	3
18	1

**Рис. 3.32.** Результат работы функции DATEDIFF

### 3.2.3.2.4. Функция преобразования типа

В тех случаях, когда Firebird не может выполнить неявное преобразование типов, требуется выполнять явное преобразование с помощью функции **CAST**. Эта функция производит преобразование значения выражения, заданного первым аргументом, в тип, заданный вторым аргументом. Синтаксис функции:

**CAST** (<выражение> AS <тип данных>).

В качестве типа данных нельзя указывать домены.

В большинстве случаев использование функции **CAST** не требуется, так как Firebird производит неявное преобразование типов данных. Например, сравнение столбца типа **DATE** с датой '12/31/2003' приведет к неявному преобразованию строкового литерала '12/31/2003' в тип данных **DATE** и следующий запрос является корректным:

```
SELECT * FROM Request WHERE IncomingDate > '01.10.2001';
```

Можно использовать функцию **CAST** для сравнения столбцов с различными типами данных из одной и той же таблицы или из различных таблиц.

С помощью **CAST** можно выполнять преобразование из одного типа дата/время в другой. В табл. 3.3 представлены правила преобразования [18].

**Таблица 3.3.** Преобразования между типами дата/время

<b>Исходный тип</b>	<b>В тип <b>TIMESTAMP</b></b>	<b>В тип <b>DATE</b></b>	<b>В тип <b>TIME</b></b>
<b>TIMESTAMP</b>	Недоступно	Да, преобразует дату, игнорируя время	Да, преобразует время, игнорируя дату
<b>DATE</b>	Да, время устанавливается в значение полуночи	Недоступно	Нет
<b>TIME</b>	Да, дате присваивается значение <b>CURRENT_DATE</b>	Нет	Недоступно
<b>DATE+TIME</b>	Да	Нет	Нет

Можно также преобразовывать правильно сформированную строку в тип «дата-время». Например, значение с типом даты из трех значений дня, месяца и года можно получить следующим образом:

```
CAST(DAY || '.' || MONTH || '.' || YEAR AS DATE),
```

где **DAY**, **MONTH** и **YEAR** могут представлять собой константы либо столбцы таблицы, в которых содержится значение дня, месяца или года соответственно.

Например, если требуется вывести различные значения месяца и года начислений за услугу газоснабжения с кодом 2, отнесенные на первое число соответствующего месяца, то запрос может выглядеть следующим образом:

```
SELECT DISTINCT NachisMonth, NachisYear,  
CAST('1.' || NachisMonth || '.' || NachisYear as date) as FirstDay
```

```
FROM NachisSumma
WHERE GazServiceCD = 2;
```

Результат выполнения запроса представлен на рис. 3.33.

NACHISLMONTH	NACHISLYEAR	FIRSTDAY
1	1999	01.01.1999
1	2000	01.01.2000
4	1999	01.04.1999
5	2001	01.05.2001
6	2001	01.06.2001
8	2001	01.08.2001
9	2000	01.09.2000
9	2001	01.09.2001
10	1998	01.10.1998
10	2001	01.10.2001
11	2001	01.11.2001
12	2000	01.12.2000
12	2001	01.12.2001

**Рис. 3.33.** Результат работы функции CAST

Кроме того, можно преобразовывать числовые типы в строку и наоборот.

Например, чтобы при выводе увеличить номер лицевого счета всех абонентов на 2, нужно использовать следующий запрос:

```
SELECT (CAST (AccountCD AS INTEGER)+2) AS New_Acc, Fio
FROM Abonent;
```

Если, например, нужно вывести значения начислений абоненту с номером лицевого счета '115705', округленными до целого значения, то для этого можно использовать следующий запрос:

```
SELECT NachisFactCD, NachisSum,
       CAST(NachisSum AS INTEGER) AS RoundSum
FROM NachisSumma
WHERE AccountCD='115705';
```

Результат выполнения запроса представлен на рис. 3.34.

NACHISLFACTCD	NACHISLSUM	ROUNDSUM
4	40,00	40
5	250,00	250
11	250,00	250
12	58,70	59
25	37,15	37
31	37,80	38
37	37,15	37
49	37,15	37

**Рис. 3.34.** Результат округления числа до целого значения с помощью функции CAST

Для получения значения, округленного до целого, можно также использовать функцию **ROUND** с указанием точности, равной нулю. В таком случае значения будут выведены с нулями после запятой.

### 3.2.3.3. Агрегатные функции

#### 3.2.3.3.1. Общее описание агрегатных функций

Для подведения итогов по данным, содержащимся в БД, в языке SQL предусмотрены агрегатные (статистические) функции. Агрегатная функция берет в качестве аргумента какой-либо столбец (для множества строк), а возвращает одно значение, определяемое типом функции:

**AVG** – среднее значение в столбце;

**SUM** – сумма значений в столбце;

**MAX** – наибольшее значение в столбце;

**MIN** – наименьшее значение в столбце;

**COUNT** – количество значений в столбце.

Аргументами агрегатных функций могут быть как столбцы таблицы, так и результаты выражений над ними. При этом выражение может быть сколь угодно сложным.

Вложенность агрегатных функций не допускается, однако из этих функций можно составлять любые выражения.

Для функций **SUM** и **AVG** столбец должен содержать числовые значения.

Специальная функция **COUNT (\*)** служит для подсчета всех без исключения строк в таблице (включая дубликаты).

Аргументу всех функций, кроме **COUNT (\*)**, может предшествовать ключевое слово **DISTINCT** (различный), указывающее, что избыточные дублирующие значения должны быть исключены перед тем, как будет применяться функция.

Если не используется предложение **GROUP BY**, но в предложении **SELECT** используется какая-либо агрегатная функция, то в качестве возвращаемых элементов *нельзя* указывать по отдельности столбцы таблиц (можно лишь в качестве аргументов агрегатных функций).

#### 3.2.3.3.2. Вычисление среднего значения

Для вычисления среднего всех значений, содержащихся в столбце, используется агрегатная функция **AVG**. Данные, содержащиеся в столбце, должны иметь числовой тип. Так как **AVG** вначале суммирует все значения, содержащиеся в столбце, а затем делит сумму на число этих значений, возвращаемый ею результат может иметь тип, не совпадающий с типом столбца. Синтаксис использования этой функции следующий:

**AVG** ({[ALL] столбец | DISTINCT столбец} | <выражение>).

При указании аргумента ALL происходит вычисление по всем значениям. Если указан аргумент DISTINCT, то перед вычислением среднего значения из рассмотрения исключаются дублирующиеся значения. Если число строк, обрабатываемых агрегатной функцией AVG, равно нулю, то функция возвращает NULL. Выражение представляет собой скалярное численное выражение языка SQL.

Например, чтобы вычислить среднее значение оплат всех абонентов, необходимо выполнить следующий запрос:

```
SELECT AVG(Paysum) FROM Paysumma;
```

Результат выполнения запроса представлен на рис. 3.35.

<b>AVG</b>
45,17

**Рис. 3.35.** Результат работы функции AVG

Аргументом агрегатной функции может быть как простое имя столбца, как в предыдущем примере, так и выражение, как, например, в следующем запросе:

```
SELECT AVG(NachisSum+2) FROM NachisSumma;
```

При вычислении среднего к каждому суммируемому значению NachisSum добавляется число 2, а затем полученная сумма делится на количество значений. В результате выполнения данного запроса будет возвращено число 47,17.

### 3.2.3.3.3. Вычисление суммы значений в столбце

Для вычисления суммы значений, содержащихся в столбце, используется агрегатная функция SUM. При этом столбец должен иметь числовой тип данных. Результат, возвращаемый этой функцией, имеет тот же тип данных, что и столбец, но количество значащих цифр может быть больше, чем количество значащих цифр отдельных значений в столбце. Использование этой функции аналогично использованию функции AVG. Синтаксис использования функции SUM следующий:

```
SUM ({[ALL] столбец | DISTINCT столбец}).
```

Например, для нахождения суммы всех значений начислений можно использовать следующий запрос:

```
SELECT SUM(NachisSum) FROM NachisSumma;
```

Результат выполнения запроса представлен на рис. 3.36.

<b>SUM</b>
2 213,61

**Рис. 3.36.** Результат работы функции SUM

### 3.2.3.3.4. Вычисление экстремумов

Для нахождения наименьшего или наибольшего значения в столбце используются агрегатные функции – соответственно MIN или MAX. При этом столбец может содержать числовые и строковые значения либо значения даты/времени. Синтаксис использования агрегатных функций по нахождению максимального и минимального значения следующий:

MAX ({[ALL] столбец | DISTINCT столбец}),

MIN ({[ALL] столбец | DISTINCT столбец}).

Результат, возвращаемый этими функциями, имеет такой же тип данных, что и сам столбец.

В случае применения функций MIN и MAX к числовым данным числа сравниваются по арифметическим правилам.

Сравнение дат происходит последовательно: более ранние значения дат считаются меньшими, чем более поздние.

Сравнение интервалов времени выполняется на основании их продолжительности: более короткие интервалы времени меньше, чем более длинные.

Например, чтобы найти в таблице PaySumma максимальное и минимальное значения оплат можно выполнить следующий запрос:

```
SELECT MAX(PaySum), MIN(PaySum) FROM PaySumma;
```

Результат выполнения запроса представлен на рис. 3.37.

MAX	MIN
250,00	8,30

Рис. 3.37. Результат работы функций MAX и MIN

### 3.2.3.3.5. Вычисление количества значений в столбце

Количество значений в столбце подсчитывает функция COUNT. При этом тип данных столбца может быть любым. Синтаксис использования этой агрегатной функции следующий:

COUNT ({ \* | [ALL] столбец | DISTINCT столбец}).

Символ звездочки в качестве аргумента функции используется для подсчета количества строк в заданной таблице, включая значения NULL. Если в качестве аргумента выступает имя столбца, то значения NULL в нем не рассматриваются.

Например, чтобы подсчитать количество строк в таблице Abonent, можно использовать следующий запрос:

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

Следующий запрос позволяет подсчитать число различных абонентов, которые подавали заявки на ремонт газового оборудования:

```
SELECT COUNT(DISTINCT AccountCD) FROM Request;
```



Результат выполнения запроса представлен на рис. 3.38.

<b>COUNT</b>
10

**Рис. 3.38.** Результат работы функции COUNT

Таким образом, только десять разных абонентов из двенадцати подавали заявки на ремонт оборудования, хотя всего в таблице ремонтных заявок содержится 23 записи о заявках (т.е. некоторые абоненты подавали заявки не один раз).

### 3.2.3.4. Функции на списке значений

#### 3.2.3.4.1. Функции MAXVALUE и MINVALUE

Функции **MAXVALUE** и **MINVALUE** возвращают максимальное и минимальное значения соответственно из списка значений своих аргументов. Имеют следующий формат:

**MAXVALUE** (<значение1> [,<значение2> ...]);

**MINVALUE** (<значение1> [,<значение2> ...]).

Например, требуется поставить в соответствие ремонтным заявкам, принятым исполнителем с кодом 1, даты их выполнения или дату 1 января 1999 года, если соответствующая заявка была выполнена раньше этой даты:

```
SELECT RequestCD,  
       MAXVALUE (ExecutionDate, CAST ('01.01.1999' AS DATE))  
FROM Request  
WHERE ExecutorCD = 1;
```

Результат выполнения запроса представлен на рис. 3.39.

Из результата предыдущего запроса следует, что заявке с кодом 3, имеющей дату выполнения 8 марта 1998 года (т.е. ранее 1 января 1999 года), поставлена в соответствие дата 1 января 1999 года.

<b>REQUESTCD</b>	<b>MAXVALUE</b>
1	20.12.2001
3	01.01.1999
6	24.06.2001
11	12.01.1999
17	06.09.2001
21	14.09.2001
22	25.05.2001

**Рис. 3.39.** Результат работы функции MAXVALUE

### 3.2.3.4.2. Функция LIST

Функция **LIST** имеет следующий формат:

`LIST ( [ {ALL | DISTINCT} ] <выражение> [ , <разделитель> ] ),`

где <разделитель> ::= { строковая\_константа | параметр | переменная }.

Функция **LIST** возвращает строку, полученную в результате соединения известных значений (не NULL) из списка, представленного набором значений аргумента функции (<выражение>). Функция возвращает NULL, если все значения из списка имеют NULL.

**Примечание.** Входные параметры и переменные могут выступать в качестве разделителей при использовании функции **LIST** в хранимых процедурах (хранимые процедуры будут подробно описаны далее).

В качестве аргумента функции **LIST** могут быть заданы числовые значения и значения типа дата/время, которые в процессе работы функции преобразуются в строку (результатирующее значение имеет тип BLOB). Следует учитывать следующие синтаксические правила:

- если ни ALL, ни DISTINCT не указаны, то по умолчанию применяется ALL;
- если <разделитель> опущен, то для разделения соединяемых величин используется запятая.

Например, для вывода в одну строку через запятую названий всех услуг газоснабжения можно использовать следующий запрос:

```
SELECT LIST (GazServiceNM) FROM Services;
```

Результат выполнения запроса представлен на рис. 3.40.

<b>LIST</b>
Доставка газа, Заявочный ремонт ГО

Рис. 3.40. Результат работы функции LIST

### 3.2.3.5. Дополнительные возможности вывода в предложении SELECT

В стандарте SQL определены средства для выбора вариантов действий в зависимости от значений данных. К таким средствам относятся операция выбора **CASE**, а также функции вывода (выбора вариантов).

В качестве функций вывода используются функции **COALESCE**, **NULLIF**, **IF** и **DECODE**, три из которых (**COALESCE**, **NULLIF** и **DECODE**) определены в стандарте SQL, а одна (**IF**) является расширением языка именно для СУБД Firebird. Функции вывода при определенных условиях фактически являются сокращенными формами операции **CASE**. Они всегда могут быть заменены эквивалентными конструкциями **CASE**, но более сложно записанными.

Операция **CASE** и функции вывода могут использоваться в списке возвращаемых столбцов предложения **SELECT**, а также в качестве элементов списка группировки предложения **GROUP BY** (будет рассмотрено позднее).

### 3.2.3.5.1. Операция выбора CASE

Операция выбора CASE позволяет определить результат в столбце TP3, исходя из определенных условий. Имеются две формы операции CASE: простая и с поиском. Простая форма имеет следующий синтаксис:

```
CASE <выражение> {WHEN <значение1> THEN результат1}
                [{WHEN <значение2> THEN результат2}] ...
                [ ELSE результат(N+1)]
```

END.

В этой форме последовательно сравниваются значения при фразах WHEN со значением заданного выражения. При первом же совпадении возвращается значение при соответствующей фразе THEN. Если совпадений нет, то возвращается результат(N+1) при фразе ELSE.

Синтаксис операции CASE с поиском имеет следующий вид:

```
CASE {WHEN <условие_поиска1> THEN результат1}
     [{ WHEN <условие_поиска2> THEN результат2}]...
     [ ELSE результат(N+1) ]
```

END.

В этом случае последовательно проверяются все условия при фразах WHEN. Если условие истинно, то возвращается результат соответствующей фразы THEN. Если ни одно условие при фразах WHEN ни оказалось истинным, возвращается результат(N+1) при фразе ELSE.

В обоих вариантах в качестве результата после фраз THEN или ELSE может быть задано либо выражение (которое может включать константы, имена столбцов, функции, а также арифметические операции и операцию конкатенации), либо NULL-значение. Если фраза ELSE отсутствует, то операция CASE при отсутствии совпадения (или истинного условия) возвращает NULL.

Рассмотрим примеры использования операции CASE в списке возвращаемых элементов предложения SELECT.

Приведем пример простой операции CASE. Пусть необходимо вывести следующую информацию о ремонтных заявках абонента, имеющего лицевой счет с номером '115705': номер заявки, номер лицевого счета абонента, сделавшего заявку, код неисправности. В зависимости от значения поля Executed необходимо вывести, погашена заявка или нет. Запрос будет выглядеть следующим образом:

```
SELECT RequestCD,
       (' Номер л/с абонента '|| AccountCD) AS Ab_Info,
       (' Код неисправности '|| FailureCD) AS Failure,
       CASE Executed
         WHEN 0 THEN 'Заявка не погашена'
         ELSE 'Погашена'
       END
FROM Request WHERE AccountCD='115705';
```

Результат выполнения запроса представлен на рис. 3.41.

REQUESTCD	AB_INFO	FAILURE	CASE
2	Номер л/с абонента 115705	Код неисправности 1	Погашена
15	Номер л/с абонента 115705	Код неисправности 5	Погашена
16	Номер л/с абонента 115705	Код неисправности 3	Заявка не погашена
17	Номер л/с абонента 115705	Код неисправности 5	Погашена
18	Номер л/с абонента 115705	Код неисправности 3	Погашена

**Рис. 3.41.** Результат выполнения запроса с простой операцией CASE

Пусть необходимо вывести информацию об оплатах со значением от 50 до 100 с указанием срока давности оплаты: если оплата была произведена до 1999 года, то вывести 'Давно', если оплата была произведена в 1999 или 2000 годах, то вывести 'Не очень давно', если позднее - 'Недавно'. Запрос с использованием операции CASE с поиском будет выглядеть следующим образом:

```
SELECT PayFactCD,
       AccountCD,
       PaySum,
       (CASE WHEN PayDate < '01.01.1999' THEN 'Давно'
            WHEN PayDate
                BETWEEN '01.01.1999' AND '31.12.2000'
                THEN 'Не очень давно'
            ELSE 'Недавно'
       END) AS Oplata
FROM PaySumma
WHERE PaySum BETWEEN 50 AND 100;
```

Результат выполнения запроса представлен на рис. 3.42.

PAYFACTCD	ACCOUNTCD	PAYSUM	OPLATA
1	005488	58,70	Недавно
3	005488	56,00	Не очень давно
7	136160	56,00	Не очень давно
9	080047	80,00	Давно
10	080047	80,00	Недавно
12	080613	56,00	Недавно
14	115705	58,70	Недавно
15	136169	58,70	Недавно
16	443069	80,00	Недавно
27	080270	57,10	Давно
29	005488	62,13	Не очень давно
37	080270	58,10	Недавно
42	080270	60,10	Недавно

**Рис. 3.42.** Результат выполнения запроса с операцией CASE с поиском

### 3.2.3.5.2. Функция COALESCE

Функция **COALESCE** используется для замены вывода неопределенного значения на вывод любого другого и имеет следующий синтаксис:

**COALESCE** (<выражение1> , <выражение2> [, <выражение3> ]...).

Эта функция имеет два или более параметров и возвращает значение первого из параметров, отличного от NULL. Функция **COALESCE** фактически представляет собой сокращение операции **CASE** и в зависимости от числа аргументов может быть заменена следующими эквивалентными ей конструкциями:

- конструкция **COALESCE** (<выражение1>, <выражение2>) эквивалентна конструкции

```
CASE WHEN <выражение1> IS NOT NULL THEN <выражение1>
      ELSE <выражение2>
END;
```

- конструкция **COALESCE** (<выражение1>, <выражение2>, ..., <выражениеN>) для  $N \geq 3$  эквивалентна конструкции

```
CASE WHEN <выражение1> IS NOT NULL THEN <выражение1>
      ELSE COALESCE (<выражение2>, ..., <выражениеN>)
END.
```

Таким образом, эквивалентная конструкция **CASE** всегда содержит в качестве условия поиска проверку соответствующего выражения из списка функции **COALESCE** на неравенство NULL.

Например, требуется вывести информацию о датах выполнения ремонтных заявок, поступивших от абонентов с номерами лицевого счетов '005488', '115705' и '080270'. Если дата выполнения неизвестна, вывести дату поступления заявки. Если ни дата поступления, ни дата выполнения не известны, то вывести 'Дата неизвестна'. Соответствующий запрос будет выглядеть следующим образом:

```
SELECT RequestCD,
       COALESCE(ExecutionDate, IncomingDate,
               'Дата не известна') AS Date_Info
FROM Request
WHERE AccountCD IN ('005488', '115705', '080270');
```

Результат выполнения запроса представлен на рис. 3.43.

Представим данный запрос с помощью операции **CASE** в следующем виде:

```
SELECT RequestCD,
       (CASE WHEN ExecutionDate IS NOT NULL
            THEN ExecutionDate
            ELSE
              CASE WHEN IncomingDate IS NOT NULL
                   THEN IncomingDate
                   ELSE 'Дата неизвестна'
              END
      )
END
```

```

END) AS Date_Info
FROM Request
WHERE AccountCD IN ('005488', '115705', '080270');

```

Результат выполнения данного запроса совпадает с результатом выполнения предыдущего запроса, использующего функцию COALESCE (рис. 3.43), однако форма записи более длинная и сложная для понимания, чем предыдущая.

REQUESTCD	DATE_INFO
1	2001-12-20
2	2001-08-12
5	2001-12-31
13	2000-12-05
14	1999-04-13
15	2000-09-23
16	2001-12-28
17	2001-09-06
18	2000-01-04
19	2001-12-27

**Рис. 3.43.** Результат выполнения запроса при использовании COALESCE

В результате для ремонтных заявок с номерами 5 и 16, у которых дата выполнения неизвестна, выведена дата поступления заявки, а для остальных заявок - дата выполнения.

### 3.2.3.5.3. Функция NULLIF

Функция **NULLIF** производит при выборке замену заданного значения на NULL. Синтаксис использования имеет следующий вид:

```
NULLIF (<выражение1>, <выражение2>),
```

где <выражение1> - столбец или вычисляемое выражение;

<выражение2> - вычисляемое выражение (может включать константы, имена столбцов, функции, а также арифметические операции и операцию конкатенации), со значением которого сравнивается значение <выражение1>.

Функция **NULLIF** возвращает NULL, если значение <выражение1> совпадает со значением <выражение2>, и значение <выражение1> в противном случае.

Использование функции **NULLIF** аналогично использованию следующей конструкции операции CASE:

```

CASE WHEN <выражение1> = <выражение2> THEN NULL
      ELSE <выражение1>

```

```
END.
```

Пусть, например, требуется вывести номера лицевого счетов, ФИО абонентов и их номера телефонов, учитывая, что номер телефона '556893' уже не существует, а новый номер неизвестен. Запрос будет выглядеть следующим образом:

```
SELECT AccountCD, Fio, NULLIF (Phone, '556893')
```

FROM Abonent;

Результат выполнения запроса представлен на рис. 3.44.

ACCOUNTCD	FIO	CASE
005488	АКСЕНОВ С.А.	<null>
115705	МИЩЕНКО Е.В.	769975
015527	КОНЮХОВ В.С.	761699
443690	ТУЛУПОВА М.И.	214833
136159	СВИРИНА З.А.	350003
443069	СТАРОДУБЦЕВ Е.В.	683014
136160	ШМАКОВ С.В.	982222
126112	МАРКОВА В.П.	683301
136169	ДЕНИСОВА Е.К.	680305
080613	ЛУКАШИНА Р.М.	254417
080047	ШУБИНА Т.П.	257842
080270	ТИМОШКИНА Н.Г.	321002

**Рис. 3.44.** Результат выполнения запроса при использовании NULLIF

Как следует из результата, у абонента, который имел номер телефона '556893', теперь номер неизвестен.

С помощью функции NULLIF можно некоторые или все значения в заданном столбце поменять на NULL, например, следующим образом:

```
SELECT AccountCD, Fio, NULLIF (Phone, Phone)
FROM Abonent;
```

Результат выполнения запроса представлен на рис. 3.45.

ACCOUNTCD	FIO	CASE
005488	АКСЕНОВ С.А.	<null>
115705	МИЩЕНКО Е.В.	<null>
015527	КОНЮХОВ В.С.	<null>
443690	ТУЛУПОВА М.И.	<null>
136159	СВИРИНА З.А.	<null>
443069	СТАРОДУБЦЕВ Е.В.	<null>
136160	ШМАКОВ С.В.	<null>
126112	МАРКОВА В.П.	<null>
136169	ДЕНИСОВА Е.К.	<null>
080613	ЛУКАШИНА Р.М.	<null>
080047	ШУБИНА Т.П.	<null>
080270	ТИМОШКИНА Н.Г.	<null>

**Рис. 3.45.** Результат выполнения запроса на замену всего столбца на NULL

#### 3.2.3.5.4. Функция IF

Две предыдущие рассмотренные функции соответствовали стандарту SQL. СУБД Firebird предоставляет возможность использования также функции IF,

которая является расширением языка и не описана в стандарте SQL. Функция имеет следующий синтаксис:

```
IIF (<условие_поиска>, <выражение1>, <выражение2>)
```

и эквивалентна следующей конструкции операции CASE:

```
CASE WHEN <условие_поиска> THEN <выражение1>  
      ELSE <выражение2>
```

```
END.
```

Данная конструкция возвращает значение <выражение1>, если <условие\_поиска> истинно, и значение <выражение2> в противном случае.

Например, необходимо вывести информацию о типах неисправностей, не детализируя неисправности газовой плиты. Запрос будет выглядеть следующим образом:

```
SELECT FailureCD,  
       IIF(FailureNM Like '%плит%',  
          'Неисправность плиты', FailureNM)  
       AS Failure_type  
FROM Disrepair;
```

Результат выполнения запроса представлен на рис. 3.46.

FAILURECD	FAILURE_TYPE
1	Засорилась водогрейная колонка
2	Не горит АГВ
3	Течет из водогрейной колонки
4	Неисправна печная горелка
5	Неисправен газовый счетчик
6	Неисправность плиты
7	Неисправность плиты
8	Неисправность плиты
12	Неизвестна

**Рис. 3.46.** Результат выполнения запроса при использовании IIF

### 3.2.3.5.5. Функция DECODE

Функция DECODE является сокращенной формой простой операции CASE и имеет следующий формат:

```
DECODE (<выражение>, <значение1>, результат1  
       [, <значение2>, результат2 ... ] [, <результат_по_умолчанию>].
```

Эта функция последовательно сравнивает значение выражения, заданного первым аргументом, с аргументами <значение1>, <значение2> и т.д. Если сравнение оказывается истинным, возвращается соответственно результат1 или результат2 и т.д. Если значение выражения не совпало ни с одним из значений из списка, то возвращается <результат\_по\_умолчанию> (аналогичен результату, задаваемому после фразы ELSE в операции CASE). Типы значения исходного



выражения и его декодированного значения (результат1, результат2 и т.д.) могут не совпадать.

Например, с помощью следующего запроса можно вывести первые 5 строк из таблицы ремонтных заявок, указав, погашены заявки или нет:

```
SELECT FIRST 5 RequestCD,  
        DECODE (Executed, 0, 'Заявка не погашена',  
                1, 'Заявка погашена', 'Неизвестно')  
FROM Request;
```

Результат выполнения запроса представлен на рис. 3.47.

REQUESTCD	CASE
1	Заявка погашена
2	Заявка погашена
3	Заявка не погашена
5	Заявка не погашена
6	Заявка погашена

Рис. 3.47. Результат работы функции DECODE

### 3.2.4. Сортировка результатов запроса

Строки ТРЗ, как и строки таблиц БД, не имеют определенного порядка. Включив в запрос SELECT предложение ORDER BY, можно отсортировать результаты запроса. Предложение ORDER BY состоит из ключевого слова ORDER BY, за которым следует список элементов сортировки, каждый из которых имеет следующий синтаксис:

```
<элемент_сортировки> ::= {[<таблица>.] столбец  
                           | порядковый_номер_столбца  
                           | псевдоним_столбца  
                           | <выражение>}  
[ASC[ENDING] | DESC[ENDING]]  
[NULLS FIRST | NULLS LAST]}... .
```

Ключевое слово DESC означает сортировку по убыванию. Если указать необязательное и используемое по умолчанию ключевое слово ASC, то сортировка будет произведена по возрастанию.

Например, для вывода начислений абонентам за декабрь 2000 года упорядоченных по убыванию значений, следует использовать следующий запрос:

```
SELECT NachislFactCD, AccountCd, NachislSum  
FROM NachislSumma  
WHERE NachislMonth=12 AND NachislYear=2000  
ORDER BY NachislSum DESC;
```

Результат выполнения запроса представлен на рис. 3.48.

NACHISLFACTCD	ACCOUNTCD	NACHISLSUM
38	080270	58,10
2	005488	46,00
50	136160	18,30

**Рис. 3.48.** Результат выполнения запроса с сортировкой по убыванию

Можно производить сортировку по нескольким столбцам. Например, требуется вывести значения оплат абонентов с лицевыми счетами '136169', '005488', '443690'. Результаты запроса упорядочить по возрастанию номеров лицевых счетов, а затем по значению оплаты. Соответствующий запрос имеет следующий вид:

```
SELECT AccountCd, PaySum
FROM PaySumma
WHERE AccountCd IN ('136169','005488','443690')
ORDER BY AccountCd, PaySum;
```

Результат выполнения запроса представлен на рис. 3.49.

ACCOUNTCD	PAYSUM
005488	46,00
005488	56,00
005488	58,70
005488	62,13
136169	20,00
136169	25,32
136169	28,32
136169	28,32
136169	58,70
443690	17,80
443690	21,67

**Рис. 3.49.** Результат выполнения запроса с сортировкой по нескольким столбцам

В списке элементов сортировки для столбцов в ТРЗ с именами и без имен (например, вычисляемых столбцов) можно указывать их *порядковый номер* в списке возвращаемых элементов предложения SELECT. Примером этому может служить следующий запрос:

```
SELECT DISTINCT AccountCD, (NachislSum +100) AS
NachislSum_100
FROM NachislSunma
WHERE GazServiceCD =2 AND AccountCD STARTING WITH '1'
ORDER BY 2;
```

Результат выполнения запроса представлен на рис. 3.50. Здесь сортировка выполняется по значениям вычисляемого столбца (NachislSum+100), который имеет порядковый номер 2 в списке возвращаемых элементов предложения SELECT.

ACCOUNTCD	NACHISLSUM_100
136160	120,00
136169	120,00
115705	140,00
136160	156,00
115705	158,70
136169	158,70
115705	350,00

**Рис. 3.50.** Результат выполнения запроса с сортировкой по номеру столбца

В предложении ORDER BY можно указать псевдоним столбца, например NachislSum\_100 в предыдущем примере. Результат выполнения будет совпадать с результатом выполнения предыдущего запроса.

Сортировку по номеру элемента в списке можно использовать при выборке всех столбцов из таблицы. Пусть, например, требуется вывести все данные об улицах, отсортированные по названию улиц. Запрос будет выглядеть следующим образом:

**SELECT \* FROM Street ORDER BY 2;**

Результат выполнения запроса представлен на рис. 3.51.

STREETCD	STREETNM
3	ВОЙКОВ ПЕРЕУЛОК
5	ГАГАРИНА УЛИЦА
7	КУТУЗОВА УЛИЦА
6	МОСКОВСКАЯ УЛИЦА
8	МОСКОВСКОЕ ШОССЕ УЛИЦА
2	НОВАЯ УЛИЦА
4	ТАТАРСКАЯ УЛИЦА
1	ЦИОЛКОВСКОГО УЛИЦА

**Рис. 3.51.** Результат выполнения запроса с сортировкой при выборке всех данных

**Примечание.** Не рекомендуется в приложениях использовать запросы с сортировкой по номерам столбцов. Это связано с тем, что со временем структура таблицы может измениться, например, в результате добавления или удаления столбцов. Как следствие, запрос с сортировкой по порядковому номеру может дать совсем другую последовательность или вообще вызвать ошибку, ссылаясь на отсутствующий столбец [22].

В качестве элемента сортировки в ORDER BY разрешено также определять *выражения*. Например, для вывода информации о первых пяти абонентах, имеющих наиболее длинные ФИО, можно использовать следующий запрос:

```
SELECT FIRST 5 * FROM Abonent
ORDER BY CHAR_LENGTH(Fio) DESC;.
```

Результат выполнения запроса представлен на рис. 3.52.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
443069	4	51	55	СТАРОДУБЦЕВ Е.В.	683014
080270	6	35	6	ТИМОШКИНА Н.Г.	321002
443690	7	5	1	ТУЛУПОВА М.И.	214833
080613	8	35	11	ЛУКАШИНА Р.М.	254417
136169	4	7	13	ДЕНИСОВА Е.К.	680305

**Рис. 3.52.** Результат выполнения запроса с сортировкой по выражению

Функция CHAR\_LENGTH, использованная в данном запросе, возвращает длину поля Fio в символах. В результате выведены данные о 5 абонентах, имеющих наиболее длинное значение поля Fio.

В соответствии с SQL-стандартом существует возможность определить положение NULL-значений в ТРЗ. Результаты могут быть отсортированы таким образом, что NULL будут располагаться выше (NULLS FIRST) или ниже (NULLS LAST) остальных результатов запроса, отличных от NULL. По умолчанию используется NULLS LAST.

**Примечание.** Если задано NULLS FIRST, то для сортировки не может быть использован индекс (использование индексов будет описано далее).

Например, для вывода номеров и дат выполнения ремонтных заявок, поданных абонентами, номера лицевого счетов которых начинаются с '08' или с '11', отсортированных по убыванию дат выполнения с NULL-значениями вначале можно использовать следующий запрос;

```
SELECT RequestCd, ExecutionDate, AccountCd
FROM REQUEST
WHERE (AccountCD LIKE '08%') OR (AccountCD LIKE '11%')
ORDER BY ExecutionDate DESC NULLS FIRST;.
```

Результат выполнения запроса представлен на рис. 3.53.

REQUESTCD	EXECUTIONDATE	ACCOUNTCD
16	<null>	115705
5	<null>	080270
19	27.12.2001	080270
20	11.10.2001	080047
17	06.09.2001	115705
2	12.08.2001	115705
6	24.06.2001	080613
15	23.09.2000	115705
18	04.01.2000	115705
7	24.10.1998	080047

**Рис. 3.53.** Результат запроса с указанием положения NULL-значений

### 3.2.5. Предложение GROUP BY

Запрос, включающий в себя предложение GROUP BY, называется запросом с группировкой, поскольку он объединяет строки исходной таблицы в группы и для каждой группы строк генерирует одну строку ТРЗ.

Элементы, указанные в предложении GROUP BY, называются *элементами группировки*, и именно они определяют, по какому признаку строки делятся на группы. При этом группой называется набор строк, имеющих одинаковое значение в элементе (элементах) группировки.

Синтаксис предложения GROUP BY имеет следующий вид:

```
GROUP BY <элемент_группировки1> [, <элемент_группировки2>]...,
```

где

```
<элемент_группировки> := {[<таблица>.] столбец  
| порядковый_номер_столбца  
| псевдоним_столбца  
| <выражение>}.
```

Фактически в качестве элемента группировки может выступать любой возвращаемый элемент, указанный в предложении SELECT, кроме значений агрегатных функций. В выражение, представляющее собой <элемент\_группировки>, могут входить скалярные функции, агрегатные функции из различных контекстов или это может быть любая CASE-операция.

Использование предложения GROUP BY имеет смысл только при наличии в списке возвращаемых элементов предложения SELECT хотя бы одного вычисляемого столбца или агрегатной функции. Агрегатная функция берет столбец значений и возвращает одно значение. Предложение GROUP BY указывает, что результаты запроса следует разделить на группы, применить агрегатную функцию по отдельности к каждой группе и получить для каждой группы одну строку результатов.

Например, если необходимо вычислить среднее значение начислений за каждый год из таблицы NachislSumma, то можно воспользоваться следующим запросом:

```
SELECT NachislYear, AVG(NachislSum) FROM NachislSumma  
GROUP BY NachislYear;
```

Результат выполнения запроса представлен на рис. 3.54.

NACHISLYEAR	AVG
1998	28,12
1999	32,89
2000	58,36
2001	58,17

Рис. 3.54. Результат выполнения запроса с группировкой

Возможна группировка результатов запроса на основании порядкового номера возвращаемого элемента в предложении SELECT. Таким образом, запрос

```
SELECT NachislYear, AVG(NachislSum) FROM NachislSumma
GROUP BY 1;
```

выдаст такой же результат, как и предыдущий запрос.

Например, если для каждого абонента требуется вывести общее количество оплат с указанным в этой же строке максимальным для него значением оплаты, то можно сгруппировать результат по номеру лицевого счета и использовать в качестве второго возвращаемого элемента выражение с агрегатными функциями COUNT и MAX. Соответствующий запрос будет выглядеть следующим образом:

```
SELECT AccountCD, (COUNT(*) || ' - с максимальной суммой '
|| MAX(PaySum)) AS Pay_Count
FROM PaySumma
GROUP BY 1;
```

Результат выполнения запроса представлен на рис. 3.55.

ACCOUNTCD	PAY_COUNT
005488	4 - с максимальной суммой 62.13
015527	3 - с максимальной суммой 38.32
080047	6 - с максимальной суммой 80.00
080270	4 - с максимальной суммой 60.10
080613	5 - с максимальной суммой 56.00
115705	8 - с максимальной суммой 250.00
126112	2 - с максимальной суммой 25.30
136159	2 - с максимальной суммой 8.30
136160	4 - с максимальной суммой 56.00
136169	5 - с максимальной суммой 58.70
443069	4 - с максимальной суммой 80.00
443690	2 - с максимальной суммой 21.67

**Рис. 3.55.** Результат выполнения запроса с группировкой

В предложении GROUP BY можно указывать псевдонимы столбцов. Например, следующий запрос позволит для каждого значения, большего 50, подсчитать количество фактов его начислений:

```
SELECT NachislSum AS Summa_50, COUNT(*)
FROM NachislSumma
WHERE NachislSum > 50
GROUP BY Summa_50;
```

Результат выполнения запроса представлен на рис. 3.56.

SUMMA_50	COUNT
56,00	3
57,10	1
58,10	1
58,70	3
60,10	1
62,13	1
80,00	3
250,00	2

**Рис. 3.56.** Результат выполнения запроса с группировкой по псевдониму

В SQL можно группировать результаты запроса *на основании двух или более элементов*.

Например, необходимо для каждого абонента вывести наименьшее значение оплаты за услуги газоснабжения за 1999 и 2000 годы. Для этого нужно сгруппировать таблицу PaySumma по номерам лицевого счетов абонентов и по годам, применяя к значениям оплат агрегатную функцию MIN следующим образом:

```
SELECT AccountCD, PayYear, MIN(Paysum)
FROM PaySumma
WHERE PayYear IN (1999, 2000)
GROUP BY AccountCD, PayYear;
```

Результат выполнения запроса представлен на рис. 3.57.

ACCOUNTCD	PAYYEAR	MIN
005488	1999	56,00
005488	2000	46,00
015527	1999	38,32
080047	1999	22,20
080270	2000	58,10
080613	2000	12,60
115705	1999	37,15
115705	2000	40,00
126112	2000	15,30
136159	1999	8,30
136160	1999	56,00
136160	2000	18,30
136169	1999	25,32
443690	1999	21,67

**Рис. 3.57.** Результат выполнения запроса с группировкой по двум столбцам

СУБД Firebird в качестве элемента группировки допускает определение *выражений*, в которых разрешено использовать SQL-функции SUBSTRING и EXTRACT, а также операцию CASE и функции вывода.

Например, требуется для каждой группы номеров лицевых счетов, начинающихся с одинаковых трех символов, вывести количество абонентов, имеющих такие счета. Запрос будет выглядеть следующим образом:

```
SELECT ('Начало счета ' ||
        SUBSTRING(AccountCD FROM 1 FOR 3)) AS Acc_3,
        COUNT(*)
FROM Abonent
GROUP BY Acc_3;
```

Результат выполнения запроса представлен на рис. 3.58.

ACC_3	COUNT
Начало счета 005	1
Начало счета 015	1
Начало счета 080	3
Начало счета 115	1
Начало счета 126	1
Начало счета 136	3
Начало счета 443	2

**Рис. 3.58.** Результат выполнения запроса с группировкой по выражению

Следует учесть, что *нельзя* выводить по отдельности столбцы, участвующие в выражении группировки.

Например, в результате попытки выполнить следующий запрос:

```
SELECT AccountCD, COUNT(*)
FROM Abonent
GROUP BY
SUBSTRING (AccountCD FROM 1 FOR 3);
```

будет выдано сообщение об ошибке, так как столбец AccountCD, участвующий в выражении группировки, присутствует в списке возвращаемых элементов предложения SELECT. Таблица разделяется на группы по первым 3 совпадающим символам номера лицевого счета, и каждый элемент из списка выбора должен иметь по одному значению для группы. В одну группу может входить несколько номеров лицевых счетов, но если попытаться вывести каждый номер лицевого счета AccountCD, будет выдано сообщение об ошибке.

Рассмотрим пример использования операции CASE в качестве элемента группировки. Например, требуется вывести средние значения начислений за годы до 2000 года и за годы после 1999 года. Соответствующий запрос будет выглядеть следующим образом:

```
SELECT 'В среднем начислено ' ||
        (CASE WHEN NachisYear < 2000
              THEN 'до 2000 года'
              ELSE 'после 1999 года'
              END) AS God,
        AVG (NachisSum) AS average_sum
FROM NachisSumma
```



## GROUP BY God;

Результат выполнения запроса представлен на рис. 3.59.

<b>GOD</b>	<b>AVERAGE_SUM</b>
В среднем начислено до 2000 года	30,40
В среднем начислено после 1999 года	58,24

**Рис. 3.59.** Результат выполнения запроса с группировкой по CASE

Как следует из предыдущих примеров, на запросы с группировкой накладываются следующие ограничения.

1. В предложении GROUP BY должны быть указаны столбцы или выражения, которые используются в качестве возвращаемых элементов предложения SELECT (за исключением агрегатных функций).
2. Все элементы списка возвращаемых столбцов должны иметь одно значение для каждой группы строк. Это означает, что возвращаемым элементом в предложении SELECT может быть:
  - константа;
  - агрегатная функция, возвращающая одно значение для всех строк, входящих в группу;
  - элемент группировки, который по определению имеет одно и то же значение во всех строках группы;
  - функция, которая используется в качестве элемента группировки;
  - выражение, включающее в себя перечисленные выше элементы.

На практике в список возвращаемых элементов запроса с группировкой всегда входят столбец или функция группировки и агрегатная функция. Если агрегатная функция не указана, значит, запрос можно более просто выразить с помощью ключевого слова *DISTINCT* без использования предложения GROUP BY. И наоборот, если не включить в результаты запроса столбец или функцию, по которым производится группировка, то станет невозможно определить, к какой группе относится каждая строка результатов.

### 3.2.6. Предложение HAVING

Предложение HAVING запроса SELECT применяется для наложения условий на строки, возвращаемые при использовании предложения GROUP BY. Оно состоит из ключевого слова HAVING, за которым следует <условие\_поиска>:

<условие\_поиска> ::= [NOT] <условие\_поиска1>  
[[AND|OR][NOT] <условие\_поиска2>]....,

где <условие\_поиска> позволяет исключить из результата группы, не удовлетворяющие заданным условиям. Условие поиска совпадает с условием

поиска, рассмотренным выше для предложения WHERE. Однако в качестве значения часто используется значение, возвращаемое агрегатными функциями.

Результат совместной работы HAVING с GROUP BY аналогичен результату работы запроса SELECT с предложением WHERE с той разницей, что HAVING выполняет те же функции над строками (группами) возвращаемого набора данных, а не над строками исходной таблицы. Из этого следует, что предложение HAVING начинает свою работу после того, как предложение GROUP BY разделит базовую таблицу на группы. В противоположность этому использование предложения WHERE приводит к тому, что сначала отбираются строки из базовой таблицы и только после этого отобранные строки начинают использоваться.

Например, чтобы для каждого из абонентов, которые подавали более одной ремонтной заявки, вывести их число и дату самой ранней из них, нужно выполнить следующий запрос:

```
SELECT AccountCD, COUNT(*), MIN(Incomingdate)
FROM Request
GROUP BY AccountCD HAVING COUNT(*) > 1;
```

Результат запроса представлен на рис. 3.60.

ACCOUNTCD	COUNT	MIN
005488	3	04.04.1999
080047	2	20.10.1998
080270	2	17.12.2001
115705	5	28.12.1999
136160	2	12.01.1999
136169	2	07.05.2001
443069	2	08.08.2001

**Рис. 3.60.** Результат выполнения запроса с HAVING

Работа этого запроса заключается в следующем. Вначале GROUP BY из таблицы Request формирует группы, состоящие из одинаковых значений поля AccountCD. После этого в предложении HAVING происходит подсчет числа строк, входящих в каждую группу, и в ТРЗ включаются все группы, которые содержат более одной строки.

Следует отметить, что если задать условие COUNT(\*)>1 в предложении WHERE, то такой запрос потерпит неудачу, так как предложение WHERE производит оценку в терминах одиночной строки, а агрегатные функции оцениваются в терминах групп строк. В то же время из этого не следует, что предложение WHERE не используется с предложением HAVING.

Следует учесть, что предложение HAVING должно ссылаться только на агрегатные функции и элементы, выбранные GROUP BY.

Например, следующий запрос потерпит неудачу:

```
SELECT AccountCD, MAX(Incomingdate) FROM Request
GROUP BY AccountCD
```

HAVING FailureCD=1;.

Поле FailureCD не может быть использовано в предложении HAVING, потому что оно может иметь (и действительно имеет) больше чем одно значение на группу вывода.

Таким образом, следующий запрос является корректным:

```
SELECT AccountCD, MAX(Incomingdate) FROM Request
WHERE FailureCD=1
GROUP BY AccountCD;.
```

Результат запроса представлен на рис. 3.61.

ACCOUNTCD	MAX
005488	17.12.2001
080270	31.12.2001
115705	07.08.2001
136169	06.11.2001

**Рис. 3.61.** Результат выполнения корректного запроса

Как отмечалось, HAVING может использовать только аргументы, которые имеют одно значение на группу вывода. **Практически ссылки на агрегатные функции – наиболее общие, но и поля (столбцы), выбранные с помощью GROUP BY, также допустимы.**

Например, если необходимо узнать максимальные значения начислений для абонентов с лицевыми счетами '005488' и '080047', то можно выполнить следующий запрос:

```
SELECT AccountCD, MAX (NachislSum)
FROM NachislSumma
GROUP BY AccountCD
HAVING AccountCD IN ('005488', '080047');.
```

Результат выполнения запроса представлен на рис. 3.62.

ACCOUNTCD	MAX
005488	62,13
080047	80,00

**Рис. 3.62.** Результат выполнения запроса

Предположим, необходимо для каждой неисправности, с которой последняя заявка поступила позднее 31.08.2001, вывести дату поступления последней заявки, общее количество заявок, а также, сколько из них было выполнено. Для решения этой задачи запрос будет выглядеть следующим образом:

```
SELECT FailureCD, MAX(IncomingDate),COUNT(*),
      'Из них выполнено ' || Count(ExecutionDate)
FROM Request
GROUP BY FailureCD
HAVING MAX(IncomingDate) > '31.08.2001';.
```

Результат выполнения запроса представлен на рис. 3.63.

FAILURECD	MAX	COUNT	CONCATENATION
1	31.12.2001	4	Из них выполнено 3
2	11.10.2001	3	Из них выполнено 3
3	28.12.2001	2	Из них выполнено 1
8	17.12.2001	2	Из них выполнено 2

**Рис. 3.63.** Результат выполнения запроса к таблице Request

Как и условие поиска в предложении WHERE, условие поиска в предложении HAVING может дать один из трех следующих результатов:

- если условие поиска имеет значение TRUE, то группа строк остается и для нее генерируется одна строка в результате запроса;
- если условие поиска имеет значение FALSE, то группа строк исключается и строка в результате запроса для нее не генерируется;
- если условие поиска имеет значение NULL, то группа строк исключается и строка в результате запроса для нее не генерируется.

Правила обработки значений NULL в условиях поиска для предложения HAVING точно такие же, как и для предложения WHERE. Предложение HAVING всегда должно использоваться в сочетании с предложением GROUP BY, хотя синтаксис запроса SELECT не требует этого.

### 3.3. Многотабличные и вложенные запросы

СУБД Firebird поддерживает три варианта запросов к множеству таблиц с помощью одного SQL-запроса на выборку: соединения, подзапросы и объединения [18, 23]. Эти три метода поиска данных во множестве таблиц существенно отличаются друг от друга и, как правило, решают различные виды поисковых задач.

**Соединение** используется в запросах SELECT для генерации наборов, содержащих столбцы из нескольких таблиц, которые хранят связанные данные. Множество столбцов, выбранных из каждой таблицы, называются *потоками*. Процесс соединения объединяет выбранные столбцы в единый выходной набор данных. Стандарты SQL поддерживают два варианта синтаксиса соединения: *неявное* и *явное соединение*.

*Неявное соединение* соответствует более старому SQL-стандарту (SQL:89). Таблицы, участвующие в соединении, задаются списком с разделяющими запятыми в предложении FROM запроса SELECT. Условия для связи таблиц задаются среди условий поиска предложения WHERE. Не существует специального синтаксиса для указания, какие условия используются для поиска, а какие – для соединения.

В стандарте SQL:92 введен более универсальный синтаксис *явного соединения*, которое осуществляется с помощью предложения JOIN. Структура предложения JOIN...ON дает возможность отличать условие соединения от

условий поиска. Следует отметить, что неявное соединение может быть всегда заменено эквивалентным явным, обратное же возможно не всегда. Различные виды явных и неявных соединений будут рассмотрены далее.

**Подзапросом** является запрос, заключенный в круглые скобки и вложенный в предложение SELECT, FROM, WHERE или HAVING основного (внешнего) запроса SELECT или *других запросов*, использующих эти предложения. Правила построения подзапросов изменяются в соответствии с целью запроса. Соединения и подзапросы используют слияние потоков данных из строк различных таблиц, поэтому их роли частично совпадают при некоторых условиях.

**Запросы объединения**, реализуемые с помощью предложения UNION, дают возможность выбрать строки из различных наборов данных в объединенный набор данных, причем подмножества не обязательно должны быть связаны друг с другом – просто они должны соответствовать друг другу структурно.

Соединения, подзапросы и объединения не являются взаимоисключающими. Соединения и объединения могут включать подзапросы, и некоторые подзапросы могут содержать соединения. Далее будут рассмотрены все виды запросов к множеству таблиц более подробно.

### 3.3.1. Соединения таблиц

#### 3.3.1.1. Неявное соединение таблиц

Как уже отмечалось, соединения используются для получения составных наборов данных, содержащих столбцы из нескольких таблиц, которые хранят связанные данные. Формат предложений FROM и WHERE при неявном соединении таблиц имеет следующий вид:

FROM <таблица1> [псевдоним1], <таблица2> [псевдоним2]...

[WHERE <условие\_соединения> [AND <условие\_поиска>]... ],

где <условие\_соединения> ::=

<таблица1>.столбец <операция\_сравнения> <таблица2>.столбец,

<операция\_сравнения> ::= { = | < | > | <= | >= | <> }.

Таким образом, особенности синтаксиса неявных соединений следующие:

- использование более одной таблицы в предложении FROM (список таблиц с разделяющими запятыми);
- среди остальных условий поиска предложения WHERE применяется операция сравнения для создания выражения, которое определяет столбцы, используемые для соединения указанных таблиц (<условие\_соединения>). При этом соединение на основе точного равенства между двумя столбцами называется *соединением по равенству*.

Алгоритм выполнения запросов на неявное соединение таблиц состоит из следующих этапов [24]:

- 1) вычисляется декартово произведение таблиц, входящих в соединение, т.е. для каждой строки одной из таблиц берутся все возможные сочетания строк из других таблиц;
- 2) производится отбор строк из полученной таблицы согласно условию поиска в предложении WHERE;
- 3) осуществляется проекция (вывод) по столбцам, указанным в списке возвращаемых элементов.

Среди запросов на соединение таблиц наиболее распространены запросы к таблицам, которые связаны с помощью *отношения родитель-потомок*. Чтобы использовать в запросе отношение родитель-потомок, необходимо задать <условие\_соединения>, в котором первичный ключ родительской таблицы сравнивается с внешним ключом таблицы-потомка (обычно имена этих столбцов совпадают в связанных таблицах). Несмотря на то, что в языке определения данных присутствует возможность декларативного задания первичных и внешних ключей таблиц, связь, о которой идет речь, должна всегда явно указываться в предложении WHERE запроса SELECT.

Например, необходимо вывести для всех абонентов названия улиц, на которых они проживают. Для этого надо каждую запись из таблицы Abonent связать по полю внешнего ключа (столбец StreetCD в таблице Abonent) с таблицей улиц (столбец StreetCD в таблице Street). Следующий запрос позволяет получить требуемый результат:

```
SELECT Abonent.Fio, Street.StreetCD, Street.Streetnm
FROM Abonent, Street
WHERE Abonent.StreetCD = Street.StreetCD;.
```

Результат выполнения запроса представлен на рис. 3.64.

<b>FIO</b>	<b>STREETCD</b>	<b>STREETNM</b>
АКСЕНОВ С.А.	3	ВОЙКОВ ПЕРЕУЛОК
МИЩЕНКО Е.В.	3	ВОЙКОВ ПЕРЕУЛОК
КОНЮХОВ В.С.	3	ВОЙКОВ ПЕРЕУЛОК
ГУЛУПОВА М.И.	7	КУТУЗОВА УЛИЦА
СВИРИНА З.А.	7	КУТУЗОВА УЛИЦА
ТИМОШКИНА Н.Г.	6	МОСКОВСКАЯ УЛИЦА
ЛУКАШИНА Р.М.	8	МОСКОВСКОЕ ШОССЕ УЛИЦА
ШУБИНА Т.П.	8	МОСКОВСКОЕ ШОССЕ УЛИЦА
СТАРДУБЦЕВ Е.В.	4	ТАТАРСКАЯ УЛИЦА
ШМАКОВ С.В.	4	ТАТАРСКАЯ УЛИЦА
МАРКОВА В.П.	4	ТАТАРСКАЯ УЛИЦА
ДЕНИСОВА Е.К.	4	ТАТАРСКАЯ УЛИЦА

**Рис. 3.64.** Результат выполнения многотабличного запроса

Следует обратить внимание на то, что к полям таблиц обращение производится на основе полного имени столбца (то есть с указанием таблицы, к которой он относится), чтобы исключить неоднозначности, возникающие в случае присутствия в разных таблицах полей с одинаковыми именами (например, полей StreetCD в таблицах Abonent и Street).

Чтобы ускорить ввод запросов и сделать их более понятными, в списке таблиц можно определить *псевдонимы таблиц* (сокращенные имена). Например, предыдущий запрос можно записать более кратко:

```
SELECT A.Fio, S.StreetCD, S.Streetnm
FROM Abonent A, Street S
WHERE A.StreetCD = S.StreetCD;
```

В многотабличном запросе можно *комбинировать условие соединения*, в котором задаются связанные столбцы (соединение с помощью равенства), с условиями поиска. Например, для вывода фамилий абонентов, которым за месяцы 2001 года начислены суммы более 50, можно использовать следующий запрос:

```
SELECT A.Fio, N.NachislSum
FROM Abonent A, NachislSumma N
WHERE A.AccountCD = N.AccountCD
AND NachislYear = 2001 AND NachislSum > 50;
```

Результат выполнения запроса представлен на рис. 3.65.

<b>FIO</b>	<b>NACHISLSUM</b>
АКСЕНОВ С.А.	58,70
МИЩЕНКО Е.В.	250,00
МИЩЕНКО Е.В.	58,70
СТАРОДУБЦЕВ Е.В.	80,00
ДЕНИСОВА Е.К.	58,70
ЛУКАШИНА Р.М.	56,00
ШУБИНА Т.П.	80,00
ТИМОШКИНА Н.Г.	60,10

**Рис. 3.65.** Результат выполнения многотабличного запроса с комбинированным условием поиска

Термин "соединение" применяется к любому запросу, который объединяет данные нескольких таблиц БД путем сравнения значений в парах столбцов этих таблиц. Самыми распространенными являются соединения, созданные на основе равенства связанных столбцов (соединения по равенству). Кроме того, имеется возможность соединять таблицы с помощью *других операций сравнения*. Например, чтобы вывести все комбинации фамилий абонентов и исполнителей ремонтных заявок так, чтобы фамилии абонентов были больше при сравнении, можно использовать следующий запрос с соединением таблиц по неравенству:

```
SELECT A.Fio, E.Fio
FROM Abonent A, Executor E
WHERE A.Fio > E.Fio;
```

Как следует из данного примера, соединения таблиц по условиям, отличающимся от равенства, во многом искусственны. Поэтому в подавляющем большинстве случаев таблицы соединяются по равенству, а

другие операции сравнения используются для дополнительного отбора строк в условии поиска.

SQL позволяет соединять *три и более таблицы*, используя ту же самую методику, что и при соединении данных из двух таблиц. Например, чтобы для предыдущего запроса вместе с начисленной суммой вывести и заплаченную сумму за тот же период и за ту же услугу газоснабжения, можно использовать следующее соединение трёх таблиц:

```
SELECT A.Fio, N.NachislSum, P.PaySum
FROM Abonent A, NachislSumma N, PaySumma P
WHERE A.AccountCD = N. AccountCD
      AND NachislYear = 2001
      AND NachislSum > 50
      AND A.AccountCD = P.AccountCD
      AND NachislMonth = PayMonth
      AND NachislYear = PayYear
      AND N.GazserviceCD = P.GazserviceCD;
```

Результат выполнения запроса представлен на рис. 3.66.

<b>FIO</b>	<b>NACHISLSUM</b>	<b>PAYSUM</b>
АКСЕНОВ С.А.	58,70	58,70
МИЩЕНКО Е.В.	250,00	250,00
МИЩЕНКО Е.В.	58,70	58,70
СТАРОДУБЦЕВ Е.В.	80,00	80,00
ДЕНИСОВА Е.К.	58,70	58,70
ЛУКАШИНА Р.М.	56,00	56,00
ШУБИНА Т.П.	80,00	80,00
ТИМОШКИНА Н.Г.	60,10	60,10

**Рис. 3.66.** Результат выполнения запроса на соединение трех таблиц

Следует отметить, что результат выполнения запроса на неявное соединение более двух таблиц не зависит от порядка перечисления этих таблиц в предложении FROM и от порядка указания условий соединения в предложении WHERE. В таблицах соединяются только те строки, для которых выполняется <условие\_соединения>, и независимо от порядка соединения результат будет одинаковый.

Можно соединять данные из трех и более таблиц, связанных более чем одним *отношением родитель-потомок*.

Например, для того чтобы определить исполнителей, которым назначены ремонтные заявки абонентов, необходимо соединить таблицы Abonent, Request и Executor. Для этого надо каждую запись из таблицы Request связать по полю внешнего ключа AccountCD со справочником абонентов (столбец AccountCD в таблице Abonent), а по полю внешнего ключа ExecutorCD – со справочником исполнителей (столбец ExecutorCD в таблице Executor). Для этого можно использовать следующий запрос:



```

SELECT DISTINCT A.Fio AS Fio_Abonent, E.Fio AS Fio_Executor
FROM Abonent A, Executor E, Request R
WHERE R.AccountCD = A.AccountCD AND
      R.ExecutorCD = E.ExecutorCD;

```

Результат выполнения запроса представлен на рис. 3.67.

<b>FIO_ABONENT</b>	<b>FIO_EXECUTOR</b>
АКСЕНОВ С.А.	СТАРОДУБЦЕВ Е.М.
АКСЕНОВ С.А.	ШКОЛЬНИКОВ С.М.
АКСЕНОВ С.А.	ШЛЮКОВ М.К.
ДЕНИСОВА Е.К.	БУЛГАКОВ Т.И.
ДЕНИСОВА Е.К.	ШКОЛЬНИКОВ С.М.
КОНЮХОВ В.С.	СТАРОДУБЦЕВ Е.М.
ЛУКАШИНА Р.М.	СТАРОДУБЦЕВ Е.М.
МИЩЕНКО Е.В.	БУЛГАКОВ Т.И.
МИЩЕНКО Е.В.	СТАРОДУБЦЕВ Е.М.
МИЩЕНКО Е.В.	ШЛЮКОВ М.К.
МИЩЕНКО Е.В.	ШУБИН В.Г.
СВИРИНА З.А.	ШУБИН В.Г.
СТАРОДУБЦЕВ Е.В.	СТАРОДУБЦЕВ Е.М.
СТАРОДУБЦЕВ Е.В.	ШКОЛЬНИКОВ С.М.
ТИМОШКИНА Н.Г.	ШЛЮКОВ М.К.
ШМАКОВ С.В.	СТАРОДУБЦЕВ Е.М.
ШУБИНА Т.П.	ШУБИН В.Г.

**Рис. 3.67.** Результат выполнения запроса на соединение трех таблиц

### 3.3.1.2. Явное соединение таблиц

Как уже было отмечено, другим способом связывания таблиц является *явное соединение*, осуществляемое с помощью предложения JOIN. Формат предложения FROM при таком соединении таблиц имеет следующий вид:

```

FROM {<таблица1> [псевдоним1] <тип_соединения1> <таблица2> [псевдоним2]
     [ { ON <условие_соединения1> | USING (<список_столбцов> ) } ]
     [<тип_соединения2> <таблица3> [псевдоним3]
     [ { ON <условие_соединения2> | USING (<список_столбцов> ) } ]}...,

```

где

```

<тип_соединения> ::= {CROSS JOIN
                     | [NATURAL] [{INNER | {LEFT | RIGHT | FULL} [OUTER]}] JOIN }.

```

Таким образом, существуют различные типы явного соединения таблиц.

1. Перекрестное соединение CROSS JOIN используется без конструкции ON <условие\_соединения>. CROSS JOIN эквивалентно декартовому произведению таблиц. То есть конструкция ... FROM A CROSS JOIN B полностью эквивалентна конструкции ... FROM A, B.

2. Уточненные соединения, которые предполагают явное задание условия соединения после ON или имен столбцов, по которым производится

соединение, после USING:

а) INNER JOIN – «внутреннее» соединение. В таблицах соединяются только те строки, для которых выполняется <условие\_соединения>;

б) OUTER JOIN – «внешнее» соединение. Данное ключевое слово является необязательным и имеет смысл только в комбинации с ключевым словом определения типа внешнего соединения. Внешнее соединение бывает трех типов:

- LEFT (OUTER) JOIN – «левое (внешнее)» соединение. Это означает, что в результат запроса будут включены все строки левой таблицы и только те строки правой таблицы, для которых выполняется <условие\_соединения>. Для строк из левой таблицы, для которых не найдено соответствия в правой таблице, в столбцы, извлекаемые из правой таблицы, заносятся значения NULL;

- RIGHT (OUTER) JOIN – «правое (внешнее)» соединение. Это означает, что в результат запроса будут включены все строки правой таблицы и только те строки левой таблицы, для которых выполняется <условие\_соединения>. Для строк из правой таблицы, для которых не найдено соответствия в левой таблице, в столбцы, извлекаемые из левой таблицы, заносятся значения NULL;

- FULL (OUTER) JOIN – «полное (внешнее)» соединение. Это комбинация «левого» и «правого» соединений таблиц.

Если не указан тип соединения в предложении JOIN, то он по умолчанию принимается за INNER.

3. Естественное соединение (доступно только для БД на диалекте 3). Стандарт SQL определяет это соединение как результат соединения таблиц по всем одноименным столбцам. Если одноименных столбцов нет, выполняется перекрестное соединение CROSS JOIN. Естественное соединение не требует задания каких-либо условий после ON. Может применяться при внутреннем и внешнем соединении таблиц.

В уточненных явных соединениях <условие\_соединения>, указываемое после ON, аналогично рассмотренному выше условию соединения в предложении WHERE. При соединении данных из нескольких таблиц конструкции JOIN и WHERE взаимозаменяемы. Однако зачастую конструкция JOIN является более удобной для понимания. Например, конструкция JOIN...ON дает возможность отличать условие соединения, указываемое после ON, от условия поиска, указываемого в предложении WHERE.

Например, необходимо вывести для всех абонентов названия и коды улиц, на которых они проживают. Решение этого примера на основе конструкции WHERE было приведено ранее (результат на рис. 3.64). Реализация этого запроса посредством явного соединения таблиц будет иметь вид:

```
SELECT A.Fio, S.StreetCD, S.StreetNM  
FROM Abonent A INNER JOIN Street S  
ON A.StreetCD = S.StreetCD;
```

Если в этом же примере использовать правое внешнее соединение, то получится список всех улиц и проживающих на них абонентов, если таковые

имеются. Запрос будет выглядеть следующим образом:

```
SELECT A.Fio, S.StreetCD, S.StreetNM  
FROM Abonent A RIGHT JOIN Street S  
ON A.StreetCD = S.StreetCD;
```

Результат выполнения запроса представлен на рис. 3.68.

<b>FIO</b>	<b>STREETCD</b>	<b>STREETNM</b>
АКСЕНОВ С.А.	3	ВОЙКОВ ПЕРЕУЛОК
МИЩЕНКО Е.В.	3	ВОЙКОВ ПЕРЕУЛОК
КОНЮХОВ В.С.	3	ВОЙКОВ ПЕРЕУЛОК
ГУЛУПОВА М.И.	7	КУТУЗОВА УЛИЦА
СВИРИНА З.А.	7	КУТУЗОВА УЛИЦА
ТИМОШКИНА Н.Г.	6	МОСКОВСКАЯ УЛИЦА
ЛУКАШИНА Р.М.	8	МОСКОВСКОЕ ШОССЕ УЛИЦА
ШУБИНА Т.П.	8	МОСКОВСКОЕ ШОССЕ УЛИЦА
СТАРОДУБЦЕВ Е.В.	4	ТАТАРСКАЯ УЛИЦА
ШМАКОВ С.В.	4	ТАТАРСКАЯ УЛИЦА
МАРКОВА В.П.	4	ТАТАРСКАЯ УЛИЦА
ДЕНИСОВА Е.К.	4	ТАТАРСКАЯ УЛИЦА
<null>	5	ГАГАРИНА УЛИЦА
<null>	1	ЦИОЛКОВСКОГО УЛИЦА
<null>	2	НОВАЯ УЛИЦА

**Рис. 3.68.** Результат правого внешнего соединения

Таким образом, для каждого абонента выводится название улицы, на которой он проживает, а если на какой-либо улице не проживают абоненты, то в поле Fio строки для данной улицы выводится значение NULL.

Если в данном примере использовать левое внешнее соединение, то результат будет совпадать с результатом внутреннего соединения (рис. 3.64), так как нет абонентов, для которых не указана улица проживания.

Если же использовать полное внешнее соединение, то для данного примера результат будет совпадать с результатом правого внешнего соединения (рис. 3.68).

Следует отметить, что необязательно в качестве имен соединяемых таблиц использовать только явные имена таблиц либо только псевдонимы таблиц. Допускается использовать также и их сочетание. Таким образом, следующий запрос вернет правильный результат:

```
SELECT Abonent.Fio, S.StreetCD, S.StreetNM  
FROM Abonent JOIN Street S  
ON Abonent.StreetCD = S.StreetCD;
```

Рассмотрим запрос, в котором используется явное соединение таблиц и условие поиска в предложении WHERE. Например, для выбора фамилий абонентов, которые производили оплату за услугу газоснабжения с кодом 2 позднее 1 октября 2001 года, такой запрос может быть построен следующим образом:

```
SELECT A.Fio, P.PaySum
```

```

FROM Abonent A JOIN PaySumma P
  ON A.AccountCD = P.AccountCD
WHERE GazServiceCD=2 AND PayDate > '01.10.2001';

```

Результат выполнения запроса представлен на рис. 3.69.

<b>ФИО</b>	<b>PAYSUM</b>
АКСЕНОВ С.А.	58,70
МИЩЕНКО Е.В.	250,00
ШУБИНА Т.П.	80,00
ТИМОШКИНА Н.Г.	46,00
ДЕНИСОВА Е.К.	58,70
СТАРОДУБЦЕВ Е.В.	80,00

**Рис. 3.69.** Результат наложения дополнительных условий поиска для соединяемых таблиц

Для БД на диалекте 3 существует возможность более простого, по сравнению с конструкцией ON, задания условия соединения – *соединение по именам столбцов*. Если таблицы соединяются по одноименным столбцам, то можно использовать конструкцию USING (<список\_столбцов>). В <списке\_столбцов> указываются имена всех столбцов, по значениям в которых требуется соединить таблицы.

При создании соединения по именам столбцов следует помнить следующее:

- все столбцы, указанные в списке столбцов, должны существовать в соединяемых таблицах;
- по всем указанным столбцам автоматически создается такое соединение, как если бы было указано  
 <таблица1>.столбец = <таблица2>.столбец  
 в предложении WHERE при неявном соединении или после ON при явном соединении.

Например, рассмотренный ранее запрос, выводящий для всех абонентов названия и коды улиц, можно реализовать с использованием конструкции USING:

```

SELECT A.Fio, S.StreetCD, S.StreetNM
FROM Abonent A INNER JOIN Street S
  USING (StreetCD);

```

Этот же запрос можно реализовать с помощью естественного соединения:

```

SELECT A.Fio, S.StreetCD, S.StreetNM
FROM Abonent A NATURAL JOIN Street S;

```

С помощью оператора JOIN можно соединять *три и более таблицы*. Порядок соединений может уточняться круглыми скобками, так как результат нескольких внешних соединений зависит от порядка их выполнения.

Например, вывести адреса и ФИО абонентов, проживающих на улицах, наименования которых начинаются с букв Г или М, указав для каждого абонента значения начислений, можно с помощью следующего запроса:

```

SELECT S.StreetNM, ('д.'||A.HouseNo) AS House,

```

```

('кв.'||A.FlatNo) AS Flat,
A.Fio,
N.NachislSum,
(N.NachislMonth || ' месяц ' || N.NachislYear || ' года') AS
Period
FROM (Abonent A RIGHT JOIN Street S
      ON A.StreetCD = S.StreetCD)
FULL JOIN NachislSumma N
      ON A.AccountCD = N.AccountCD
WHERE (S.StreetNM LIKE 'М%') OR (S.StreetNM LIKE 'Г%')
ORDER BY N.NachislYear DESC, N.NachislMonth DESC;

```

Результат выполнения запроса представлен на рис. 3.70.

STREETNM	HOUSE	FLAT	FIO	NACHISLSUM	PERIOD
МОСКОВСКАЯ УЛИЦА	д.35	кв.6	ТИМОШКИНА Н.Г.	46,00	12 месяц 2001 года
МОСКОВСКОЕ ШОССЕ УЛИЦА	д.39	кв.36	ШУБИНА Т.П.	80,00	10 месяц 2001 года
МОСКОВСКОЕ ШОССЕ УЛИЦА	д.39	кв.36	ШУБИНА Т.П.	32,56	9 месяц 2001 года
МОСКОВСКОЕ ШОССЕ УЛИЦА	д.35	кв.11	ЛУКАШИНА Р.М.	56,00	6 месяц 2001 года
МОСКОВСКАЯ УЛИЦА	д.35	кв.6	ТИМОШКИНА Н.Г.	60,10	5 месяц 2001 года
МОСКОВСКАЯ УЛИЦА	д.35	кв.6	ТИМОШКИНА Н.Г.	58,10	12 месяц 2000 года
МОСКОВСКОЕ ШОССЕ УЛИЦА	д.35	кв.11	ЛУКАШИНА Р.М.	12,60	8 месяц 2000 года
МОСКОВСКОЕ ШОССЕ УЛИЦА	д.35	кв.11	ЛУКАШИНА Р.М.	22,86	4 месяц 2000 года
МОСКОВСКОЕ ШОССЕ УЛИЦА	д.39	кв.36	ШУБИНА Т.П.	22,20	7 месяц 1999 года
МОСКОВСКОЕ ШОССЕ УЛИЦА	д.39	кв.36	ШУБИНА Т.П.	22,56	5 месяц 1999 года
МОСКОВСКОЕ ШОССЕ УЛИЦА	д.39	кв.36	ШУБИНА Т.П.	80,00	10 месяц 1998 года
МОСКОВСКОЕ ШОССЕ УЛИЦА	д.35	кв.11	ЛУКАШИНА Р.М.	10,60	9 месяц 1998 года
МОСКОВСКОЕ ШОССЕ УЛИЦА	д.35	кв.11	ЛУКАШИНА Р.М.	12,60	4 месяц 1998 года
МОСКОВСКОЕ ШОССЕ УЛИЦА	д.39	кв.36	ШУБИНА Т.П.	19,56	3 месяц 1998 года
МОСКОВСКАЯ УЛИЦА	д.35	кв.6	ТИМОШКИНА Н.Г.	57,10	2 месяц 1998 года
ГАГАРИНА УЛИЦА	<null>	<null>	<null>	<null>	<null>

**Рис. 3.70.** Результат соединения трех таблиц

В ТРЗ включена и ГАГАРИНА УЛИЦА, на которой не проживают абоненты, так как сначала выполнялось правое внешнее соединение таблиц Abonent и Street, а затем полное внешнее соединение полученной таблицы с таблицей NachislSumma. Итоговый результат отсортирован по убыванию года начислений, а внутри года – по убыванию месяца.

### 3.3.1.3. Стандартные соединения (объединения) таблиц

#### 3.3.1.3.1. Декартово произведение

Для получения декартова произведения таблиц в предложении FROM необходимо указать перечень перемножаемых таблиц, а в предложении SELECT – все их столбцы. Перемножим таблицы Abonent (12 строк) и Street (8 строк) и получим результирующую таблицу (96 строк):

```

SELECT Abonent.*, Street.*
FROM Abonent, Street;

```

Фрагмент результата выполнения запроса представлен на рис. 3.71.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE	STREETCD1	STREETNM
005488	3	4	1	АКСЕНОВ С.А.	556893	3	ВОЙКОВ ПЕРЕУЛОК
115705	3	1	82	МИЩЕНКО Е.В.	769975	3	ВОЙКОВ ПЕРЕУЛОК
015527	3	1	65	КОНЮХОВ В.С.	761699	3	ВОЙКОВ ПЕРЕУЛОК
...	...	...	...	...	...	...	...
005488	3	4	1	АКСЕНОВ С.А.	556893	7	КУТУЗОВА УЛИЦА
115705	3	1	82	МИЩЕНКО Е.В.	769975	7	КУТУЗОВА УЛИЦА
015527	3	1	65	КОНЮХОВ В.С.	761699	7	КУТУЗОВА УЛИЦА
...	...	...	...	...	...	...	...
080047	8	39	36	ШУБИНА Т.П.	257842	2	НОВАЯ УЛИЦА
080270	6	35	6	ГИМОШКИНА Н.Г.	321002	2	НОВАЯ УЛИЦА

**Рис. 3.71.** Декартово произведение таблиц Abonent и Street

Такой же результат может быть получен, если использовать следующий запрос на явное соединение:

```
SELECT Abonent.*, Street.*
FROM Abonent CROSS JOIN Street;
```

Таким образом, декартово произведение двух таблиц – это набор всевозможных комбинаций строк из двух таблиц.

### 3.3.1.3.2. Эквисоединение

Для получения эквисоединения таблиц необходимо для декартова произведения таблиц установить имеющее смысл соответствие на основе равенства между столбцами соединяемых таблиц.

Например, запрос на неявное эквисоединение таблиц Abonent и Street будет выглядеть следующим образом:

```
SELECT Abonent.*, Street.*
FROM Abonent, Street
WHERE Abonent.StreetCD = Street.StreetCD;
```

Фрагмент результата выполнения запроса представлен на рис. 3.72.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE	STREETCD1	STREETNM
005488	3	4	1	АКСЕНОВ С.А.	556893	3	ВОЙКОВ ПЕРЕУЛОК
115705	3	1	82	МИЩЕНКО Е.В.	769975	3	ВОЙКОВ ПЕРЕУЛОК
015527	3	1	65	КОНЮХОВ В.С.	761699	3	ВОЙКОВ ПЕРЕУЛОК
443690	7	5	1	ТУЛУПОВА М.И.	214833	7	КУТУЗОВА УЛИЦА
136159	7	39	1	СВИРИНА З.А.	350003	7	КУТУЗОВА УЛИЦА
...	...	...	...	...	...	...	...

**Рис. 3.72.** Эквисоединение таблиц Abonent и Street

Такой же результат может быть получен, если использовать следующий запрос на явное соединение:

```
SELECT Abonent.*, Street.*
FROM Abonent JOIN Street
ON Abonent.StreetCD = Street.StreetCD;
```

### 3.3.1.3.3. Естественное соединение таблиц

Для получения естественного соединения таблиц необходимо в эквисоединении таблиц исключить дубликаты повторяющихся столбцов (столбцов, входящих в условие соединения). Для предыдущего примера естественное соединение таблиц Abonent и Street по столбцу StreetCD выглядит следующим образом:

```
SELECT AccountCD, Street.StreetCD, StreetNM, HouseNo, FlatNo,
       Fio, Phone
FROM Abonent, Street
WHERE Abonent.StreetCD = Street.StreetCD;
```

Фрагмент результата выполнения запроса представлен на рис. 3.73.

ACCOUNTCD	STREETCD	STREETNM	HOUSENO	FLATNO	FIO	PHONE
005488	3	ВОЙКОВ ПЕРЕУЛОК	4	1	АКСЕНОВ С.А.	556893
115705	3	ВОЙКОВ ПЕРЕУЛОК	1	82	МИЩЕНКО Е.В.	769975
015527	3	ВОЙКОВ ПЕРЕУЛОК	1	65	КОНЮХОВ В.С.	761699
443690	7	КУТУЗОВА УЛИЦА	5	1	ТУЛУПОВА М.И.	214833
136159	7	КУТУЗОВА УЛИЦА	39	1	СВИРИНА З.А.	350003
080270	6	МОСКОВСКАЯ УЛИЦА	35	6	ТИМОШКИНА Н.Г.	321002
...	...	...	...	...	...	...

**Рис. 3.73.** Естественное соединение таблиц Abonent и Street

Такой же результат может быть получен, если использовать следующий запрос на явное естественное соединение:

```
SELECT AccountCD, StreetCD, StreetNM, HouseNo, FlatNo, Fio, Phone
FROM Abonent NATURAL JOIN Street;
```

### 3.3.1.3.4. Композиция

Для создания композиции таблиц нужно исключить из вывода все столбцы, по которым проводилось соединение таблиц, например следующим образом:

```
SELECT AccountCD, StreetNM, HouseNo, FlatNo, Fio, Phone
FROM Abonent, Street
WHERE Abonent.StreetCD = Street.StreetCD;
```

Фрагмент результата выполнения запроса представлен на рис. 3.74.

ACCOUNTCD	STREETNM	HOUSENO	FLATNO	FIO	PHONE
005488	ВОЙКОВ ПЕРЕУЛОК	4	1	АКСЕНОВ С.А.	556893
115705	ВОЙКОВ ПЕРЕУЛОК	1	82	МИЩЕНКО Е.В.	769975
015527	ВОЙКОВ ПЕРЕУЛОК	1	65	КОНЮХОВ В.С.	761699
443690	КУТУЗОВА УЛИЦА	5	1	ГУЛУПОВА М.И.	214833
136159	КУТУЗОВА УЛИЦА	39	1	СВИРИНА З.А.	350003
080270	МОСКОВСКАЯ УЛИЦА	35	6	ТИМОШКИНА Н.Г.	321002
...	...	...	...	...	...

**Рис. 3.74.** Композиция таблиц Abonent и Street

### 3.3.1.3.5. Тета-соединение

Тета-соединение предназначено для тех случаев, когда необходимо соединить две таблицы на основе некоторых условий, отличных от равенства.

Например, получить тета-соединение таблиц Abonent и Street можно следующим образом:

```
SELECT Abonent.*, Street.*
FROM Abonent, Street
WHERE Abonent.StreetCD < Street.StreetCD;
```

Фрагмент результата выполнения запроса представлен на рис. 3.75.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE	STREETCD1	STREETNM
005488	3	4	1	АКСЕНОВ С.А.	556893	7	КУТУЗОВА УЛИЦА
115705	3	1	82	МИЩЕНКО Е.В.	769975	7	КУТУЗОВА УЛИЦА
015527	3	1	65	КОНЮХОВ В.С.	761699	7	КУТУЗОВА УЛИЦА
443069	4	51	55	СТАРДУБЦЕВ Е.В.	683014	7	КУТУЗОВА УЛИЦА
136160	4	9	15	ШМАКОВ С.В.	982222	7	КУТУЗОВА УЛИЦА
126112	4	7	11	МАРКОВА В.П.	683301	7	КУТУЗОВА УЛИЦА
136169	4	7	13	ДЕНИСОВА Е.К.	680305	7	КУТУЗОВА УЛИЦА
080270	6	35	6	ТИМОШКИНА Н.Г.	321002	7	КУТУЗОВА УЛИЦА
...	...	...	...	...	...	...	...
015527	3	1	65	КОНЮХОВ В.С.	761699	5	ГАГАРИНА УЛИЦА
443069	4	51	55	СТАРДУБЦЕВ Е.В.	683014	5	ГАГАРИНА УЛИЦА
136160	4	9	15	ШМАКОВ С.В.	982222	5	ГАГАРИНА УЛИЦА
126112	4	7	11	МАРКОВА В.П.	683301	5	ГАГАРИНА УЛИЦА
136169	4	7	13	ДЕНИСОВА Е.К.	680305	5	ГАГАРИНА УЛИЦА

**Рис. 3.75.** Тета-соединение таблиц Abonent и Street



### 3.3.1.4. Соединение таблицы со своей копией

Некоторые многотабличные запросы используют отношения, существующие внутри одной из таблиц. Чтобы обратиться к одной и той же таблице внутри одного запроса, используется псевдоним таблицы, определяемый непосредственно после имени таблицы в предложении FROM запроса SELECT. Например, чтобы найти все пары абонентов, проживающих на одной и той же улице, можно использовать следующее неявное соединение таблицы Abonent со своей копией:

```
SELECT F.Fio, S.Fio
FROM Abonent F, Abonent S
WHERE F.StreetCD = S.StreetCD AND F.Fio < S.Fio;.
```

В этом примере для таблицы Abonent определены два псевдонима: F (First) и S (Second). Эти псевдонимы будут существовать, пока выполняется запрос. Дополнительное условие поиска "F.Fio < S.Fio" предназначено для удаления из ТРЗ повторяющихся строк, появляющихся в результате того, что запрос выбирает все комбинации строк с одинаковым кодом улицы.

Результат выполнения запроса представлен на рис. 3.76.

<b>FIO</b>	<b>FIO1</b>
АКСЕНОВ С.А.	МИЩЕНКО Е.В.
АКСЕНОВ С.А.	КОНЮХОВ В.С.
КОНЮХОВ В.С.	МИЩЕНКО Е.В.
СВИРИНА З.А.	ГУЛУПОВА М.И.
СТАРОДУБЦЕВ Е.В.	ШМАКОВ С.В.
МАРКОВА В.П.	СТАРОДУБЦЕВ Е.В.
МАРКОВА В.П.	ШМАКОВ С.В.
ДЕНИСОВА Е.К.	СТАРОДУБЦЕВ Е.В.
ДЕНИСОВА Е.К.	ШМАКОВ С.В.
ДЕНИСОВА Е.К.	МАРКОВА В.П.
ЛУКАШИНА Р.М.	ШУБИНА Т.П.

**Рис. 3.76.** Результат соединения таблицы со своей копией

Такой же результат может быть получен, если использовать следующий запрос на *явное соединение*:

```
SELECT F.Fio, S.Fio
FROM Abonent F JOIN Abonent S ON F.StreetCD = S.StreetCD
WHERE F.Fio < S.Fio;.
```

Примером *соединения таблицы со своей копией и другой таблицей* может быть следующий запрос, выводящий все пары абонентов, имеющих ремонтные заявки с одной и той же неисправностью газового оборудования:

```
SELECT FailureNM, A.AccountCD, B.AccountCD
FROM Request A, Request B, Disrepair D
WHERE A.FailureCD = B.FailureCD AND D.FailureCD = A.FailureCD
AND A.AccountCD < B.AccountCD;.
```

Результат выполнения запроса представлен на рис. 3.77.

FAILURENM	ACCOUNTCD	ACCOUNTCD1
Засорилась водогрейная колонка	005488	115705
Засорилась водогрейная колонка	005488	080270
Засорилась водогрейная колонка	005488	136169
Засорилась водогрейная колонка	115705	136169
Засорилась водогрейная колонка	080270	115705
Засорилась водогрейная колонка	080270	136169
Не горит АГВ	080047	443069
Не горит АГВ	080047	443069
Плохое поступление газа на горелку плиты	080613	136160
Плохое поступление газа на горелку плиты	005488	080613
Плохое поступление газа на горелку плиты	005488	136160
Туго поворачивается пробка крана плиты	136160	136169
При закрытии краника горелка плиты не гаснет	005488	080270
Неизвестна	015527	136159

**Рис. 3.77.** Результат соединения таблицы со своей копией и другой таблицей

### 3.3.2. Запросы с вложенными запросами

#### 3.3.2.1. Виды вложенных запросов

Вложенный запрос - это запрос, заключенный в круглые скобки и вложенный в предложение WITH, SELECT, FROM, WHERE или HAVING основного (внешнего) запроса SELECT или *других запросов*, использующих эти предложения. Вложенный запрос также называют подзапросом. Вложенный запрос в своих предложениях может содержать другой вложенный запрос и т.д.

Условно подзапросы подразделяют на три типа, каждый из которых является сужением предыдущего:

<табличный\_подзапрос> ::= запрос SELECT, возвращающий набор строк и столбцов;

<подзапрос\_столбца> ::= запрос SELECT, возвращающий значения только одного столбца, но, возможно, в нескольких строках;

<скалярный\_подзапрос> ::= запрос SELECT, возвращающий значение одного столбца в одной строке.

Использование подзапросов в предложении WITH имеет следующий вид:

```
WITH [RECURSIVE]
  имя_производной_таблицы1 [(<список_столбцов>)]
  AS (<табличный_подзапрос>)
  [, имя_производной_таблицы2 [(<список_столбцов>)]
  AS (<табличный_подзапрос>)]....
```

При использовании вложенных запросов в предложении SELECT синтаксис возвращаемых элементов имеет следующий вид:

```
<возвращаемый_элемент> ::=  
{ [<таблица>].* | [<таблица>.] столбец | константа | <выражение>  
| (<скалярный_подзапрос>)} .
```

При использовании вложенных запросов в предложении FROM его синтаксис имеет следующий вид:

```
FROM <производная_таблица1> [, <производная_таблица2> ]...,  
где <производная_таблица> ::=  
(<табличный_подзапрос>) [[AS] псевдоним] [(<список_столбцов>)] .
```

При использовании вложенных запросов в предложениях WHERE и HAVING изменяется синтаксис некоторых условий поиска. Простое сравнение при использовании вложенного запроса реализуется следующей конструкцией:

```
<значение> <операция_сравнения> { <значение1>  
| (<скалярный_подзапрос>)  
| {ANY| ALL} (<подзапрос_столбца>)} .
```

Проверка на членство в множестве реализуется следующей конструкцией:

```
<значение> [NOT] IN ({<значение1> [, <значение2> ...] | <подзапрос_столбца>} ) .
```

Также при использовании вложенных запросов есть возможность осуществлять проверку на существование с помощью предиката EXISTS и проверку на возврат подзапросом единственного значения с помощью предиката SINGULAR. Условие поиска с проверкой существования представляется в следующем виде:

```
[NOT] EXISTS (<табличный_подзапрос>).
```

Условие поиска с проверкой на единственное возвращаемое значение представляется в следующем виде:

```
SINGULAR (<табличный_подзапрос>).
```

Использование конструкций FIRST, SKIP, PLAN, ORDER BY и ROWS разрешено для любого подзапроса.

Существуют простые и соотнесенные (связанные) вложенные запросы. В предложениях SELECT, WHERE и HAVING могут использоваться и простые, и соотнесенные вложенные запросы, а в предложении FROM только простые вложенные запросы. В предложении WHERE (или HAVING) как простые, так и соотнесенные вложенные запросы включаются с помощью предикатов IN, EXISTS или одной из операций сравнения ( = | <> | < | <= | > | >= ). Следует отметить, что выражения, содержащие подзапрос в предложениях WHERE или HAVING, используются наиболее часто.

*Простым вложенным запросом* называется такой, результат которого не зависит от внешнего запроса. Данные из таблиц, указанных в предложениях FROM внешнего запроса и подзапроса, извлекаются независимо друг от друга, вследствие чего необязательно вводить псевдонимы для этих таблиц или указывать полные имена столбцов. Простые вложенные запросы

обрабатываются системой "снизу- вверх". Первым обрабатывается вложенный запрос самого нижнего уровня. Множество значений, полученное в результате его выполнения, используется при реализации запроса более высокого уровня и т.д.

*Связанным вложенным запросом* называется такой, результат которого зависит от результата внешнего запроса. Подзапрос является связанным, когда в нем (в предложениях WHERE, HAVING) указан столбец таблицы внешнего запроса. Такое обращение к столбцам внешнего запроса называется внешней ссылкой. Если быть точнее, внешняя ссылка – это имя столбца одной из таблиц, указанных в предложении FROM внешнего запроса, но не входящего ни в одну из таблиц предложения FROM подзапроса. В связанных подзапросах следует указывать полные имена столбцов, причем если во внешнем и вложенном запросах используется одна и та же таблица, то для столбцов должны быть заданы псевдонимы. Запросы со связанными вложенными запросами обрабатываются в обратном порядке ("сверху-вниз"), т.е. сначала выбирается первая строка рабочей таблицы, сформированная основным запросом. Затем из нее выбираются значения тех столбцов, которые используются в подзапросе (подзапросах). Если эти значения удовлетворяют условиям вложенного запроса, то выбранная строка включается в результат. После этого во внешнем запросе выбирается вторая строка и т.д., пока в результат не будут включены все строки, удовлетворяющие подзапросу (последовательности подзапросов).

### **3.3.2.2. Запросы с простыми подзапросами**

#### **3.3.2.2.1. Простые подзапросы в предложении WITH**

Производные таблицы, возвращаемые табличным подзапросом, могут быть определены в предложении WITH, которое записывается перед основным запросом SELECT.

Синтаксис использования предложения WITH следующий:

```
WITH [RECURSIVE]
  имя_производной_таблицы1 [(<список_столбцов>)]
  AS (<табличный_подзапрос> )
  [, имя_производной_таблицы2 [(<список_столбцов>)]
  AS (<табличный_подзапрос>)]....
```

Как следует из приведенного синтаксиса, в предложении WITH может быть определено несколько подзапросов. Сами предложения WITH не могут быть вложенными.

Подзапросы в предложении WITH могут быть нерекурсивными и рекурсивными. Если используется рекурсивный подзапрос, то в предложении WITH указывается ключевое слово RECURSIVE.

В качестве табличного подзапроса может использоваться любой запрос SELECT, причем при использовании рекурсии <табличный\_подзапрос> обязательно содержит в себе объединение результатов нескольких запросов.

Рассмотрим пример использования нерекурсивной производной таблицы, определенной в предложении WITH. Допустим, необходимо вывести по каждому абоненту номер его лицевого счета, ФИО и общие суммы оплат за 2000 и 2001 годы. Запрос будет выглядеть следующим образом:

```
WITH Year_Abon_Pay AS
(SELECT PayYear, AccountCD, SUM(PaySum) AS Total_Sum
FROM PaySumma GROUP BY PayYear, AccountCD)
SELECT A.AccountCD, A.Fio,
      God_2000.Total_Sum AS Total_2000,
      God_2001.Total_Sum AS Total_2001
FROM Abonent A
LEFT JOIN Year_Abon_Pay God_2000
ON A.AccountCD = God_2000.AccountCD
   AND God_2000.PayYear = 2000
LEFT JOIN Year_Abon_Pay God_2001
ON A.AccountCD = God_2001.AccountCD
   AND God_2001.PayYear = 2001;
```

Результат выполнения запроса представлен на рис. 3.78.

ACCOUNTCD	ФИО	TOTAL_2000	TOTAL_2001
005488	АКСЕНОВ С.А.	108,13	58,70
115705	МИЩЕНКО Е.В.	290,00	346,50
015527	КОНЮХОВ В.С.	<null>	<null>
443690	ТУЛУПОВА М.И.	<null>	<null>
136159	СВИРИНА З.А.	<null>	<null>
443069	СТАРОДУБЦЕВ Е.В.	<null>	156,82
136160	ШМАКОВ С.В.	18,30	20,00
126112	МАРКОВА В.П.	15,30	25,30
136169	ДЕНИСОВА Е.К.	<null>	107,02
080613	ЛУКАШИНА Р.М.	35,46	56,00
080047	ШУБИНА Т.П.	<null>	112,56
080270	ТИМОШКИНА Н.Г.	58,10	106,10

**Рис. 3.78.** Результат выполнения запроса с нерекурсивной производной таблицей

Следует отметить следующие особенности использования нерекурсивных подзапросов в предложении WITH:

- производные таблицы, определенные в предложении WITH, могут ссылаться друг на друга;
- ссылка на производную таблицу (имя\_производной\_таблицы) может использоваться в любой части основного запроса (в предложениях SELECT, FROM и т.д.);

- одна и та же производная таблица может использоваться несколько раз в основном запросе под разными псевдонимами;
- в многострочных запросах на обновление (INSERT, UPDATE и DELETE) подзапросы могут включать предложение WITH, определяющее производные таблицы;
- производные таблицы могут использоваться в процедурном языке.

Рекурсивные производные таблицы в предложении WITH позволяют создавать рекурсивные запросы.

Рассмотрим пример использования рекурсивной производной таблицы, определенной в предложении WITH. Предположим, что в таблицу Abonent был добавлен еще один столбец HEAD\_ACCOUNT (это можно сделать с помощью запроса ALTER TABLE, использование которого будет рассмотрено позднее). В этом столбце для каждого абонента указан лицевой счет управляющего по дому, в котором проживает абонент. Если абонент сам является управляющим, то в столбце HEAD\_ACCOUNT указывается NULL. Допустим, в таблице Abonent имеются данные, представленные на рис. 3.79.

ACCOUNTCD	HEAD_ACCOUNT	STREETCD	HOUSENO	FLATNO	FIO	PHONE
005488	<null>	3	4	1	АКСЕНОВ С.А.	556893
015527	115705	3	1	65	КОНЮХОВ В.С.	761699
080047	<null>	8	39	36	ШУБИНА Т.П.	257842
080270	<null>	6	35	6	ТИМОШКИНА Н.Г.	321002
080613	<null>	8	35	11	ЛУКАШИНА Р.М.	254417
115705	<null>	3	1	82	МИЩЕНКО Е.В.	769975
126112	136169	4	7	11	МАРКОВА В.П.	683301
136159	<null>	7	39	1	СВИРИНА З.А.	350003
136160	<null>	4	9	15	ШМАКОВ С.В.	982222
136169	<null>	4	7	13	ДЕНИСОВА Е.К.	680305
443069	<null>	4	51	55	СТАРОДУБЦЕВ Е.В.	683014
443690	<null>	7	5	1	ГУЛУПОВА М.И.	214833

**Рис. 3.79.** Данные таблицы Abonent с дополнительным столбцом

Допустим, необходимо вывести ту же информацию, что и в предыдущем примере, но вывод представить в виде дерева: перед ФИО каждого управляющего по дому поставить '+', а ниже вывести всех абонентов, проживающих в данном доме, с указанием перед их ФИО 4 символов пробела. Запрос будет выглядеть следующим образом:

```
WITH RECURSIVE
Year_Abon_Pay AS
  (SELECT PayYear, AccountCD, SUM(PaySum) AS Total_Sum
   FROM PaySumma GROUP BY PayYear, AccountCD),
Abonent_Tree AS
  (SELECT AccountCD, Head_AccountCD, Fio,
         CAST ('+' AS VARCHAR(4)) AS Indent
   FROM Abonent
   WHERE Head_AccountCD IS NULL
```

```

UNION ALL
SELECT A.AccountCD, A.Head_AccountCD, A.Fio,
      CAST (' ' AS VARCHAR(4)) AS Indent
FROM Abonent A JOIN Abonent_Tree Tr
      ON A.Head_AccountCD = Tr.AccountCD)
SELECT A.AccountCD, A.Indent || A.Fio AS Fio,
      God_2000.Total_Sum AS Total_2000,
      God_2001.Total_Sum AS Total_2001
FROM Abonent_Tree A
LEFT JOIN Year_Abon_Pay God_2000
      ON A.AccountCD = God_2000.AccountCD
      AND God_2000.PayYear = 2000
LEFT JOIN Year_Abon_Pay God_2001
      ON A.AccountCD = God_2001.AccountCD
      AND God_2001.PayYear = 2001;.

```

Результат выполнения запроса представлен на рис. 3.80.

ACCOUNTCD	FIO	TOTAL_2000	TOTAL_2001
005488	+АКСЕНОВ С.А.	108,13	58,70
115705	+МИЩЕНКО Е.В.	290,00	346,50
015527	КОНЮХОВ В.С.	<null>	<null>
443690	+ТУЛУПОВА М.И.	<null>	<null>
136159	+СВИРИНА З.А.	<null>	<null>
443069	+СТАРОДУБЦЕВ Е.В.	<null>	156,82
136160	+ШМАКОВ С.В.	18,30	20,00
136169	+ДЕНИСОВА Е.К.	<null>	107,02
126112	МАРКОВА В.П.	15,30	25,30
080613	+ЛУКАШИНА Р.М.	35,46	56,00
080047	+ШУБИНА Т.П.	<null>	112,56
080270	+ТИМОШКИНА Н.Г.	58,10	106,10

**Рис. 3.80.** Результат выполнения запроса с рекурсией

Необходимо указать следующие особенности использования рекурсивных производных таблиц:

- рекурсивная производная таблица имеет ссылку на саму себя;
- рекурсивная производная таблица – это объединение в одном запросе (UNION) рекурсивных и нерекурсивных частей;
- должна присутствовать, по крайней мере, одна нерекурсивная часть;
- нерекурсивные части располагаются в начале запроса, содержащего объединение;
- рекурсивные части отделяются от нерекурсивных и от самих себя с помощью конструкции UNION ALL;
- использование предложений DISTINCT, GROUP BY, HAVING, а также использование агрегатных функций не допускается в рекурсивных частях запроса;

- рекурсивная часть может иметь только одну ссылку на саму себя и только в предложении FROM;
- рекурсивная ссылка не может участвовать во внешнем соединении таблиц.

Запросы с рекурсивными производными таблицами выполняются следующим образом: выбирается первая строка из нерекурсивной части запроса; для данной строки выполняется каждая рекурсивная часть с учетом текущих значений строки как параметров; если текущая рекурсивная часть не возвращает строк, происходит возврат на шаг назад и получение следующей строки из результирующего набора нерекурсивной части запроса.

### 3.3.2.2. Простые подзапросы в предложении SELECT

Как уже отмечалось, в предложении SELECT могут использоваться простые и соотнесенные вложенные запросы.

При использовании простого подзапроса возвращенный им результат вставляется во все строки, формируемые внешним запросом. В предложении SELECT может использоваться только <скалярный\_подзапрос>, то есть подзапрос, который возвращает только одно значение.

Например, необходимо по каждому абоненту вывести среднее значение его оплат, а также среднее значение начислений по всем абонентам. Запрос может выглядеть следующим образом:

```
SELECT AccountCD, AVG (PaySum) AS AVG_Pay,
      (SELECT AVG (NachislSum) FROM NachislSumma )
      AS AVG_All_Nachisl
FROM PaySumma
GROUP BY AccountCD;
```

Результат выполнения запроса представлен на рис. 3.81.

ACCOUNTCD	AVG_PAY	AVG_ALL_NACHISL
005488	55,70	45,17
015527	28,32	45,17
080047	42,81	45,17
080270	55,32	45,17
080613	22,93	45,17
115705	93,49	45,17
126112	20,30	45,17
136159	8,30	45,17
136160	28,15	45,17
136169	32,13	45,17
443069	48,77	45,17
443690	19,73	45,17

**Рис. 3.81.** Результат выполнения вложенного запроса



Как следует из этого примера, связь между значением, возвращаемым простым вложенным запросом (среднее значение всех начисленных сумм по всем абонентам), и значениями внешнего запроса фактически отсутствует. Поэтому простые подзапросы в предложении SELECT используются достаточно редко. Область применения связанных вложенных запросов в предложении SELECT намного шире и будет рассмотрена далее.

### 3.3.2.2.3. Простые подзапросы в предложении FROM

В соответствии со стандартом SQL2003 в предложении FROM могут быть определены не базовые таблицы, а производные таблицы, возвращаемые вложенным запросом (<табличный\_подзапрос>). Производные таблицы могут быть вложенными друг в друга и могут быть включены в соединение (неявное или явное) как обычные таблицы или представления.

Следует учесть, что:

- для определения производных таблиц можно использовать только простые подзапросы;
- каждый столбец в производной таблице должен иметь имя. Если в качестве возвращаемого элемента в производной таблице используется константа, то для такого столбца должен быть введен псевдоним или должны указываться имена столбцов с помощью следующей конструкции:  
[AS] псевдоним (<список\_столбцов>);
- если используется конструкция [AS] псевдоним (<список\_столбцов>), то количество столбцов в скобках должно быть таким же, как и количество столбцов в предложении SELECT подзапроса.

Рассмотрим простейшее определение производной таблицы с помощью следующего подзапроса:

```
SELECT *  
FROM (SELECT AccountCD, Fio, Phone FROM Abonent)  
AS A (ID, Full_name, Tel);,
```

где А – псевдоним производной таблицы, а ID, Full\_name, Tel – список ее столбцов.

Данный пример иллюстрирует правила использования подзапроса в предложении FROM, но использовать его нецелесообразно, так как выборку тех же значений можно получить, используя следующий обычный запрос:

```
SELECT AccountCD, Fio, Phone FROM Abonent;
```

Рассмотрим варианты, когда использование подзапроса в предложении FROM может оказаться более полезным.

В предложении FROM могут быть определены две и более производные таблицы. Например, требуется вывести среднее количество ремонтных заявок, приходящихся на одного абонента. Для этого нужно определить общее количество ремонтных заявок, общее число абонентов и поделить полученное количество заявок на число абонентов. Запрос будет выглядеть следующим образом:

```

SELECT (CAST (R.Req_Count AS NUMERIC(5,2)) / A.Ab_Count)
      AS Req_on_Ab
FROM (SELECT COUNT (*) FROM Abonent) AS A (Ab_Count),
     (SELECT COUNT (*) FROM Request) AS R (Req_Count);

```

Результат выполнения запроса представлен на рис. 3.82.

REQ_ON_AB
1,75

**Рис. 3.82.** Результат выполнения вложенного запроса в предложении FROM

В настоящем примере используется функция CAST для преобразования вычисленного целого значения (Req\_Count) в десятичный формат NUMERIC(5,2). Если не сделать такое преобразование, то при делении целого числа на целое (R.Req\_Count/A.Ab\_Count) произойдет округление результата до целого в меньшую сторону (будет выведен результат Req\_on\_Ab=1).

Следует учесть, что предыдущий запрос выдаст ошибку деления на ноль, если таблица Abonent будет пустой. Чтобы исключить такую ошибку, можно записать предыдущий запрос в следующем виде:

```

SELECT IIF(A.Ab_Count > 0,
          (CAST (R.Req_Count AS NUMERIC(5,2))/ A.Ab_Count),
          'Нет ни одного абонента') AS Req_on_Ab
FROM (SELECT COUNT (*) FROM Abonent)
     AS A (Ab_Count),
     (SELECT COUNT (*) FROM Request)
     AS R (Req_Count);

```

В данном случае деление производится, только если количество абонентов больше нуля, в противном случае выводится 'Нет ни одного абонента'.

Рассмотрим более сложный пример вложенного запроса, когда производная таблица в предложении FROM получается путем соединения двух таблиц. Пусть требуется вывести информацию о том, сколько абонентов подали одинаковое количество ремонтных заявок, и число этих заявок. Запрос на вывод требуемой информации имеет следующий вид:

```

SELECT (Count (*)||' абонентов подали '||Req_Count
      ||' заявки') AS Info
FROM (SELECT A.AccountCD, Count (*)
      FROM Abonent A JOIN Request R
           ON A.AccountCD = R.AccountCD
      GROUP BY AccountCD)
     AS Ar (Abonent_ID, Req_Count)
GROUP BY Ar.Req_Count;

```

Результат выполнения запроса представлен на рис. 3.83.

INFO
3 абонентов подали 1 заявки
5 абонентов подали 2 заявки
1 абонентов подали 3 заявки
1 абонентов подали 5 заявки

**Рис. 3.83.** Результат выполнения вложенного запроса в предложении FROM с соединением

В этом запросе сначала формируется производная таблица Ar, которая содержит информацию о номере лицевого счета абонента Abonent\_ID и количестве поданных этим абонентом заявок Req\_Count.

Вид производной таблицы Ar представлен на рис. 3.84.

ABONENT_ID	REQ_COUNT
005488	3
015527	1
080047	2
080270	2
080613	1
115705	5
136159	1
136160	2
136169	2
443069	2

**Рис. 3.84.** Вид производной таблицы

Затем внешний запрос группирует строки из этой таблицы по количеству поданных заявок, подсчитывает количество строк в каждой группе и выводит данные в два столбца. Таким образом, получено, что одну ремонтную заявку подали три абонента, две заявки – пять абонентов и т. д. (рис. 3.83).

Допускается определение производной таблицы без использования конструкции

[AS] псевдоним (<список\_столбцов>).

Например, предыдущий пример может быть записан следующим образом:

```
SELECT (Count (*)||' абонентов подали '||Req_Count
      ||' заявки') AS Info
FROM (SELECT A.AccountCD, Count (*) AS Req_Count
      FROM Abonent A JOIN Request R
      ON A.AccountCD = R.AccountCD
      GROUP BY AccountCD)
GROUP BY Req_Count;
```

Результат выполнения запроса будет совпадать с результатом, представленным на рис. 3.83.

### 3.3.2.2.4. Простые подзапросы в предложениях WHERE и HAVING

Наиболее часто вложенные запросы используются в условиях поиска предложений WHERE и HAVING. В зависимости от того, в каком условии поиска используется подзапрос, он может представлять собой <скалярный\_подзапрос>, <подзапрос\_столбца> или <табличный\_подзапрос>.

При простом сравнении используется <скалярный\_подзапрос> либо <подзапрос\_столбца>, если перед ним указан предикат ANY или ALL. Также <подзапрос\_столбца> используется при проверке на членство во множестве. В условии поиска с предикатом EXISTS или с предикатом SINGULAR используется <табличный\_подзапрос>. Использование подзапросов с предикатами ANY, ALL и EXISTS, SINGULAR будет рассмотрено позднее после изучения простых и связанных подзапросов.

Рассмотрим использование простых подзапросов в условиях поиска предложений WHERE и HAVING.

Предположим, что известно ФИО абонента ШМАКОВ С.В., но не известно значение его поля AccountCD. Необходимо извлечь из таблицы NachislSumma все данные о начислениях абоненту ШМАКОВ С.В. Ниже приведен простой вложенный запрос, извлекающий требуемый результат из таблицы NachislSumma:

```
SELECT *
FROM NachislSumma
WHERE AccountCD = (SELECT AccountCD FROM Abonent
                   WHERE Fio = 'ШМАКОВ С.В.')
ORDER BY NachislFactCD;
```

Результат выполнения запроса представлен на рис. 3.85.

NACHISLFACTCD	ACCOUNTCD	GAZSERVICECD	NACHISLSUM	NACHISLMONTH	NACHISLYEAR
1	136160	2	56,00	1	1999
6	136160	1	18,30	1	1998
13	136160	2	20,00	5	2001
50	136160	1	18,30	12	2000

**Рис. 3.85.** Результат выполнения вложенного запроса в предложении WHERE

В данном примере подзапрос в условии поиска представляет собой <скалярный\_подзапрос>. Он выполняется первым и возвращает единственное значение поля AccountCD = 136160. Оно помещается в условие поиска основного (внешнего) запроса так, что условие поиска будет выглядеть следующим образом: "WHERE AccountCD = 136160".

**Примечание.** В предложении SELECT вложенный запрос должен выбрать одно и только одно значение, а тип данных этого значения должен совпадать с типом того значения, с которым он будет сравниваться в основном запросе.

Запрос предыдущего примера вернет во всех столбцах TP3 NULL-значения, если в таблице Abonent не будет абонента с ФИО ШМАКОВ С.В. Вложенные запросы, которые не производят никакого вывода (нулевой вывод), вынуждают рассматривать результат не как верный, не как неверный, а как неизвестный. Однако неизвестный результат имеет тот же самый эффект, что и неверный: никакие строки не выбираются основным запросом.

Запрос предыдущего примера не выполнится, если в таблице Abonent будет более одного абонента с ФИО ШМАКОВ С.В., т.к. вложенный запрос вернет более одного значения.

Например, следующий запрос, который должен найти абонентов, имеющих погашенные заявки на ремонт газового оборудования, не может быть выполнен из-за ошибки "multiple rows in singleton select" (многочисленные строки в единичном запросе SELECT):

```
SELECT * FROM Abonent WHERE AccountCD =
(SELECT AccountCD FROM Request
WHERE Executed=1 GROUP BY AccountCD);
```

Это происходит потому, что вложенный запрос возвращает более одного значения. Если в БД будет одно значение или его вообще не будет, то запрос выполнится нормально, а если несколько, то возникнет ошибка.

Для обработки множества значений, возвращаемых вложенным запросом, следует использовать специальный предикат *IN*. Тогда приведенный выше запрос может быть правильно реализован в следующем виде:

```
SELECT *
FROM Abonent
WHERE AccountCD IN
(SELECT AccountCD FROM Request
WHERE Executed = 1 GROUP BY AccountCD);
```

Результат выполнения запроса представлен на рис. 3.86.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	ФИО	PHONE
005488	3	4	1	АКСЕНОВ С.А.	556893
115705	3	1	82	МИЩЕНКО Е.В.	769975
443069	4	51	55	СТАРОДУБЦЕВ Е.В.	683014
136160	4	9	15	ШМАКОВ С.В.	982222
136169	4	7	13	ДЕНИСОВА Е.К.	680305
080613	8	35	11	ЛУКАШИНА Р.М.	254417
080047	8	39	36	ШУБИНА Т.П.	257842
080270	6	35	6	ТИМОШКИНА Н.Г.	321002

**Рис. 3.86.** Результат выполнения вложенного запроса с предикатом *IN*

В данном примере подзапрос в условии поиска представляет собой <подзапрос\_столбца>, возвращающий различные значения поля AccountCD (005488, 080047, 080270, 080613 и т.д.), где Executed = 1. Затем выполняется внешний запрос, выводящий те строки из таблицы Abonent, для которых верно условие поиска "AccountCD IN (005488, 080047, 080270, 080613 и т.д.)". Таким

образом, если используется предикат IN, то вложенный запрос выполняется один раз и формирует множество значений, используемых основным запросом. В любой ситуации, где используется реляционная операция сравнения (=), разрешается использовать IN. В отличие от запроса со знаком равенства, запрос с предикатом IN не потерпит неудачу, если больше чем одно значение выбрано вложенным запросом.

Следует отметить, что с помощью предыдущего запроса получены данные об абонентах, которые имеют погашенные ремонтные заявки, но этот запрос не дает информации об абонентах, все заявки которых погашены. Чтобы получить данные об абонентах, все заявки которых погашены, предыдущий запрос можно модифицировать следующим образом:

```
SELECT * FROM Abonent
WHERE AccountCD IN
  (SELECT AccountCD FROM Request
   WHERE Executed = 1 GROUP BY AccountCD)
AND AccountCD NOT IN
  (SELECT AccountCD FROM Request
   WHERE Executed = 0 GROUP BY AccountCD);
```

Результат выполнения этого запроса представлен на рис. 3.87. Абонентов с номерами лицевых счетов '115705' и '080270' нет в результирующей таблице, так как у этих абонентов имеются непогашенные заявки.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
005488	3	4	1	АКСЕНОВ С.А.	556893
443069	4	51	55	СТАРОДУБЦЕВ Е.В.	683014
136160	4	9	15	ШМАКОВ С.В.	982222
136169	4	7	13	ДЕНИСОВА Е.К.	680305
080613	8	35	11	ЛУКАШИНА Р.М.	254417
080047	8	39	36	ШУБИНА Т.П.	257842

**Рис. 3.87.** Результат выполнения запроса с двумя вложенными запросами

Во вложенном запросе возможно использование *той же таблицы, что и в основном запросе*. Например, если требуется вывести все данные об абоненте АКСЕНОВ С.А. и обо всех других абонентах, которые проживают с ним на одной улице, то запрос может иметь следующий вид:

```
SELECT *
FROM Abonent
WHERE StreetCD = (SELECT StreetCD FROM Abonent
                  WHERE Fio='АКСЕНОВ С.А.');
```

Результат выполнения запроса представлен на рис. 3.88.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
005488	3	4	1	АКСЕНОВ С.А.	556893
115705	3	1	82	МИЩЕНКО Е.В.	769975
015527	3	1	65	КОНЮХОВ В.С.	761699

**Рис. 3.88.** Результат использования одной и той же таблицы

В данном примере подзапрос выполняется независимо от внешнего запроса, так как является простым. Сначала будет выполнен вложенный запрос, который выберет значение StreetCD из таблицы Abonent для абонента АКСЕНОВ С.А. Затем основной запрос выберет из той же таблицы Abonent строки со значением поля StreetCD, равным значению, выбранному вложенным запросом.

Во вложенном запросе можно использовать *агрегатные функции*. Допустим, необходимо вывести абонентов и значения их начислений за 2001 год, превышающие среднюю сумму начислений по всем абонентам за этот год. Запрос будет иметь следующий вид:

```
SELECT AccountCD, NachisSum, NachisMonth, NachisYear,
       (SELECT AVG(NachisSum)
        FROM NachisSumma
        GROUP BY NachisYear
        HAVING NachisYear=2001) AS Avg_All
FROM NachisSumma
WHERE NachisSum > (SELECT AVG(NachisSum)
                   FROM NachisSumma
                   GROUP BY NachisYear
                   HAVING NachisYear=2001)
AND NachisYear=2001
ORDER BY 1;
```

Результат выполнения запроса представлен на рис. 3.89.

ACCOUNTCD	NACHISLUM	NACHISLMONTH	NACHISLYEAR	AVG_ALL
005488	58,70	12	2001	58,17
080047	80,00	10	2001	58,17
080270	60,10	5	2001	58,17
115705	250,00	9	2001	58,17
115705	58,70	8	2001	58,17
136169	58,70	11	2001	58,17
443069	80,00	9	2001	58,17

**Рис. 3.89.** Результат использования одной и той же таблицы

В этом примере вложенный запрос выполняется один раз, возвращая среднее значение поля NachisSum за 2001 год. Затем это значение последовательно сравнивается с каждой выбираемой строкой из таблицы NachisSumma. Рассмотрим еще два примера. Для вывода погашенных ремонтных заявок с наиболее поздней датой поступления можно, используя следующий запрос:

```

SELECT * FROM Request
WHERE IncomingDate = (SELECT MAX (IncomingDate)
FROM Request
WHERE Executed=1);

```

Результат выполнения запроса представлен на рис. 3.90.

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
1	005488	1	1	17.12.2001	20.12.2001	1
19	080270	4	8	17.12.2001	27.12.2001	1

**Рис. 3.90.** Результат использования агрегатной функции

Для вывода абонентов с наибольшими значениями оплаты можно использовать следующий запрос:

```

SELECT Abonent.Fio, PaySumma.*
FROM PaySumma, Abonent
WHERE PaySum = (SELECT MAX(PaySum) FROM PaySumma)
AND Abonent.AccountCD = PaySumma.AccountCD;

```

Результат выполнения запроса представлен на рис. 3.91.

ФИО	PAYFACTCD	ACCOUNTCD	GAZSERVICECD	PAYSUM	PAYDATE	PAYMONTH	PAYYEAR
МИЩЕНКО Е.В.	5	115705	2	250,00	03.10.2001	9	2001
МИЩЕНКО Е.В.	13	115705	2	250,00	06.10.2000	9	2000

**Рис. 3.91.** Результат выполнения запроса

Вложенные запросы можно применять в предложении *HAVING*. Они могут использовать свои собственные агрегатные функции (если эти функции не возвращают многочисленных значений). Также в подзапросе, включенном в условие поиска предложения *HAVING* внешнего запроса, могут использоваться свои собственные предложения *GROUP BY* и *HAVING*. Следует помнить, что аргументы, указанные в *HAVING*, должны присутствовать в качестве аргументов и в *GROUP BY*.

Например, для подсчёта числа абонентов с максимальными значениями оплаты за 2000 год можно использовать следующий запрос:

```

SELECT COUNT(DISTINCT AccountCD), PaySum
FROM PaySumma
GROUP BY PaySum
HAVING PaySum = (SELECT MAX(PaySum)
FROM PaySumma WHERE PayYear = 2000);

```

Результат выполнения запроса представлен на рис. 3.92.

COUNT	PAYSUM
1	250,00

**Рис. 3.92.** Результат вложенного запроса в предложении *HAVING*



### 3.3.2.3. Запросы со связанными подзапросами

Вложенный запрос может ссылаться на таблицу, указанную во внешнем (основном) запросе (независимо от его уровня вложенности). Такой вложенный запрос называется соотнесенным или связанным из-за того, что его результат зависит от значений, определенных в основном запросе. При этом вложенный запрос выполняется неоднократно, по одному разу для каждой строки таблицы основного (внешнего) запроса, а не один раз, как в случае простого вложенного запроса. Строка внешнего запроса, для которой внутренний запрос каждый раз будет выполнен, называется *текущей строкой-кандидатом*.

Процедура оценки, выполняемой при использовании связанного вложенного запроса, состоит из следующих шагов.

1. Выбрать строку из таблицы, именованной во внешнем запросе. Это будет *текущая строка-кандидат*.
2. Сохранить значение из этой строки-кандидата в псевдониме, который задан в предложении FROM внешнего запроса.
3. Выполнить вложенный запрос. Везде, где псевдоним, данный для внешнего запроса, найден, использовать значение для текущей строки-кандидата. Использование значения из строки-кандидата внешнего запроса во вложенном запросе называется *внешней ссылкой*.
4. Если связанный подзапрос используется в предложении WHERE или HAVING, то оценить условие поиска внешнего запроса на основе результатов вложенного запроса, выполняемого на шаге 3. Он определяет, выбирается ли строка-кандидат для вывода. Если связанный подзапрос используется в предложении SELECT, то выводятся поля, указанные в списке возвращаемых элементов основного запроса, и результат выполнения вложенного запроса.
5. Повторить процедуру для следующей строки-кандидата основной (внешней) таблицы и так далее, пока все строки таблицы не будут проверены.

Таким образом, хотя общая структура связанного подзапроса такая же, как и простого подзапроса (употребляются те же самые предложения, порядок их следования не меняется), однако в предложении WHERE или HAVING связанного подзапроса содержится ссылка на столбец таблицы внешнего запроса, и алгоритм выполнения связанного подзапроса совершенно другой.

Так как вложенный запрос содержит ссылки на таблицу (таблицы) основного запроса, то вероятность неоднозначных ссылок на имена столбцов достаточно высока. Поэтому если во вложенном запросе присутствует неполное имя столбца, то сервер БД должен определить, относится ли оно к таблице, указанной в предложении FROM самого вложенного запроса, или к таблице, указанной в предложении FROM внешнего запроса, содержащего данный вложенный запрос.

Возможные неоднозначности при определении столбца устраняются использованием полного имени столбца.

Двусмысленность при определении таблицы, используемой для конкретного отбора строк, устраняется с помощью псевдонимов таблиц, указываемых во внешнем и внутреннем запросе.

### 3.3.2.3.1. Связанные подзапросы в предложении SELECT

Чаще всего в предложении SELECT применяются связанные вложенные, а не простые запросы.

Связанный вложенный запрос, возвращающий фамилии абонентов и названия улиц, на которых они проживают, имеет вид:

```
SELECT A.Fio, (SELECT S.StreetNM FROM Street S
              WHERE S.StreetCD = A.StreetCD) AS StreetNM
FROM Abonent A;
```

Результат выполнения запроса представлен на рис. 3.93.

ФИО	STREETNM
АКСЕНОВ С.А.	ВОЙКОВ ПЕРЕУЛОК
МИЩЕНКО Е.В.	ВОЙКОВ ПЕРЕУЛОК
КОНЮХОВ В.С.	ВОЙКОВ ПЕРЕУЛОК
ТУЛУПОВА М.И.	КУТУЗОВА УЛИЦА
СВИРИНА З.А.	КУТУЗОВА УЛИЦА
СТАРОДУБЦЕВ Е.В.	ТАТАРСКАЯ УЛИЦА
ШМАКОВ С.В.	ТАТАРСКАЯ УЛИЦА
МАРКОВА В.П.	ТАТАРСКАЯ УЛИЦА
ДЕНИСОВА Е.К.	ТАТАРСКАЯ УЛИЦА
ЛУКАШИНА Р.М.	МОСКОВСКОЕ ШОССЕ УЛИЦА
ШУБИНА Т.П.	МОСКОВСКОЕ ШОССЕ УЛИЦА
ТИМОШКИНА Н.Г.	МОСКОВСКАЯ УЛИЦА

**Рис. 3.93.** Результат выполнения связанного подзапроса в предложении SELECT

В соответствии с алгоритмом, описанным выше, данный запрос работает следующим образом.

1. Внешний запрос выбирает из таблицы Abonent строку с данными об абоненте, проживающем на улице с кодом, равным 3 (первая строка).
2. Сохраняет эту строку как текущую строку-кандидат под псевдонимом А.
3. Выполняет вложенный запрос, просматривающий всю таблицу Street, чтобы найти строку, где значение поля S.StreetCD такое же, как значение A.StreetCD (3). Из найденной строки таблицы Street извлекается поле StreetNM.
4. Для вывода выбираются значение поля A.Fio из основного запроса (АКСЕНОВ С.А.) и найденное значение поля S.StreetNM из вложенного запроса (ВОЙКОВ ПЕРЕУЛОК).

5. Повторяются пп.1-4, пока каждая строка таблицы Abonent не будет проверена.

Следует отметить, что ту же задачу можно решить с использованием следующего неявного соединения таблиц Abonent и Street:

```
SELECT A.Fio, S.StreetNM
FROM Abonent A, Street S
WHERE A.StreetCD = S.StreetCD,;
```

или следующего явного соединения этих же таблиц:

```
SELECT A.Fio, S.StreetNM
FROM Abonent A INNER JOIN Street S ON
A.StreetCD = S.StreetCD,;
```

Пусть необходимо вывести для каждого абонента номер его лицевого счета, ФИО и общее количество поданных им заявок. Для этого может использоваться следующий запрос:

```
SELECT A.AccountCD, A.Fio,
       (SELECT COUNT (*) FROM Request R
        WHERE A.AccountCD = R.AccountCD)
       AS Request_Count
FROM Abonent A,;
```

Результат выполнения запроса представлен на рис. 3.94.

ACCOUNTCD	FIO	REQUEST_COUNT
005488	АКСЕНОВ С.А.	3
115705	МИЩЕНКО Е.В.	5
015527	КОНЮХОВ В.С.	1
443690	ТУЛУПОВА М.И.	0
136159	СВИРИНА З.А.	1
443069	СТАРОДУБЦЕВ Е.В.	2
136160	ШМАКОВ С.В.	2
126112	МАРКОВА В.П.	0
136169	ДЕНИСОВА Е.К.	2
080613	ЛУКАШИНА Р.М.	1
080047	ШУБИНА Т.П.	2
080270	ТИМОШКИНА Н.Г.	2

**Рис. 3.94.** Результат соотнесенного вложенного запроса

Данный запрос работает следующим образом.

1. Внешний запрос из таблицы Abonent выбирает строку с данными об абоненте, имеющем номер лицевого счета '005488' (первая строка).
2. Сохраняет эту строку как текущую строку-кандидат под псевдонимом А.
3. Выполняет вложенный запрос, просматривающий всю таблицу Request, чтобы найти все строки, где значение поля R.AccountCD такое же, как значение A.AccountCD (005488). С помощью агрегатной функции COUNT (3) подсчитывается общее количество таких строк.

4. Для вывода выбираются значения полей A.AccountCD и A.Fio из основного запроса ('005488', 'АКСЕНОВ С.А.') и найденное вложенным запросом количество связанных строк в таблице Request (3).
5. Повторяются пп.1-4, пока каждая строка таблицы Abonent не будет просмотрена.

Если во внешнем запросе используется предложение GROUP BY, то выражения, указанные в нем, можно использовать внутри подзапросов.

Например, с помощью следующего связанного вложенного запроса можно получить общие суммы начислений и оплат по каждому абоненту:

```
SELECT AccountCD,
       (SELECT Sum (NachislSum) FROM NachislSumma N
        WHERE N.AccountCD = P.AccountCD)
       AS Nachisl,
       Sum (PaySum) AS Pay
FROM PaySumma P
GROUP BY AccountCD;
```

Результат выполнения запроса представлен на рис. 3.95.

ACCOUNTCD	NACHISL	PAY
005488	222,83	222,83
015527	84,96	84,96
080047	256,88	256,88
080270	221,30	221,30
080613	114,66	114,66
115705	747,95	747,95
126112	40,60	40,60
136159	16,60	16,60
136160	112,60	112,60
136169	160,66	160,66
443069	195,10	195,10
443690	39,47	39,47

**Рис. 3.95.** Результат соотнесенного вложенного запроса

Здесь в подзапросе вычисляется общая сумма всех начислений для абонента, отобранного внешним запросом. Затем возвращенное подзапросом значение выводится по каждому абоненту в результирующем столбце Nachisl. Из результата следует, что каждый абонент полностью оплатил начисленные ему суммы.

### 3.3.2.3.2. Связанные подзапросы в предложениях WHERE и HAVING

При использовании связанного вложенного запроса в условиях поиска предложений WHERE и HAVING он может представлять собой <скалярный\_подзапрос>, <подзапрос\_столбца> или <табличный\_подзапрос>, как и для простых вложенных запросов. Однако так как запрос связанный,

внутренний запрос выполняется отдельно для каждой строки внешнего запроса (*текущая строка-кандидат*).

Рассмотрим примеры, в которых используются <скалярный\_подзапрос> и <подзапрос\_столбца>. Подзапросы, представляющие собой <табличный\_подзапрос>, будут рассмотрены позднее при изучении предикатов EXISTS и SINGULAR.

Например, чтобы вывести все сведения об абонентах, которые подали заявки на ремонт газового оборудования 17 декабря 2001 года, можно использовать следующий связанный вложенный запрос:

```
SELECT * FROM Abonent Out
WHERE '17.12.2001' IN
(SELECT IncomingDate FROM Request Inn
WHERE Out. AccountCD = Inn. AccountCD);
```

Результат выполнения запроса представлен на рис. 3.96.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
005488	3	4	1	АКСЕНОВ С.А.	556893
080270	6	35	6	ТИМОШКИНА Н.Г.	321002

**Рис. 3.96.** Результат связанного вложенного запроса в предложении WHERE

В этом примере Out и Inn - это псевдонимы таблиц Abonent и Request соответственно (могут задаваться произвольно). Так как значение в поле AccountCD внешнего запроса меняется (при переборе строк), внутренний запрос должен выполняться отдельно для каждой строки внешнего запроса.

В этом примере SQL осуществляет следующую процедуру:

- 1) выбирает строку с данными об абоненте, имеющем номер лицевого счета '005488' (первая строка) из таблицы Abonent;
- 2) сохраняет эту строку как текущую строку-кандидат под псевдонимом Out;
- 3) выполняет вложенный запрос, просматривающий всю таблицу Request, чтобы найти строки, где значение поля Inn.AccountCD - такое же, как значение Out.AccountCD (005488). Затем *из каждой такой строки таблицы Request* извлекается поле IncomingDate. В результате вложенный запрос, представляющий собой <подзапрос\_столбца>, формирует набор значений поля IncomingDate для текущей строки-кандидата;
- 4) после получения набора всех значений поля IncomingDate для поля AccountCD='005488' анализируется условие поиска основного запроса, чтобы проверить, имеется ли значение 17 декабря 2001 в наборе всех значений поля IncomingDate. Если это так (а это так), то выбирается строка с номером лицевого счета '005488' для вывода ее из основного запроса;
- 5) повторяются пп.1-4 (для строки с номером лицевого счета '015527' и т.д.), пока каждая строка таблицы Abonent не будет проверена.

Ту же самую задачу можно решить, используя естественное соединение таблиц Abonent и Request следующим образом:

```
SELECT AccountCD, StreetCD, HouseNO, FlatNO, Fio, Phone  
FROM Abonent out NATURAL JOIN Request inn  
WHERE inn.IncomingDate = '17.12.2001';
```

Результат выполнения будет совпадать с результатом, представленным на рис. 3.96. Однако следует обратить внимание на наличие существенных различий между соединением таблиц и вложенными соотнесенными запросами. Дело в том, что запросы с использованием соединения таблиц формируются СУБД как строки из декартова произведения таблиц, перечисленных в предложении FROM. В случае же с вложенным соотнесенным запросом строки из произведения таблиц не вычисляются благодаря использованию механизма строки-кандидата. Вывод в связанном вложенном запросе формируется в предложении SELECT внешнего запроса, в то время как соединения могут выводить строки из обеих соединяемых таблиц (при указании символа \* в предложении SELECT). Но даже если столбцы для вывода при соединении таблиц указаны явно (см. предыдущий пример), то сначала все равно формируется декартово произведение.

Каждый SQL-запрос можно оценить с точки зрения используемых ресурсов сервера БД. На практике большинство СУБД подзапросы выполняют более эффективно. Тем не менее, при проектировании комплекса программ с критичными требованиями по быстродействию разработчик должен проанализировать план выполнения SQL-запроса для конкретной СУБД. Тестирование в реальных условиях – единственный надежный способ решить, что лучше работает для конкретных потребностей.

Рассмотрим пример сравнения значения, возвращаемого вложенным запросом, с константой. Вывести информацию об исполнителях ремонтных заявок, назначенных на выполнение четырех и более заявок, можно с помощью следующего запроса:

```
SELECT *  
FROM Executor E  
WHERE 4<= (SELECT COUNT(R.RequestCD) FROM Request R  
          WHERE E.ExecutorCD = R.ExecutorCD);
```

Результат выполнения запроса представлен на рис. 3.97.

EXECUTORCD	FIO
1	СТАРОДУБЦЕВ Е.М.
3	ШУБИН В.Г.
4	ШЛЮКОВ М.К.

**Рис. 3.97.** Результат сравнения вложенного запроса с константой

В данном примере связанный подзапрос в условии поиска представляет собой <скалярный\_подзапрос>. Он возвращает одно единственное значение (количество ремонтных заявок) для текущей строки-кандидата, выбранной из таблицы Executor. Если это значение больше или равно 4, то текущая строка-кандидат выбирается для вывода из основного запроса. Эта процедура повторяется, пока каждая строка таблицы Executor не будет проверена.

В SQL имеется возможность использовать соотнесенный вложенный запрос, основанный на той же самой таблице, что и основной запрос. Это позволяет использовать соотнесенные вложенные запросы для извлечения сложных форм производной информации. Например, вывести размеры начислений, превышающие среднее значение начислений для каждого абонента, можно с помощью следующего запроса:

```
SELECT F.NachislSum,
       (SELECT AVG(D.NachislSum)
        FROM NachislSumma D
        WHERE F.AccountCD = D.AccountCD) AS AVG_D,
       A.AccountCD, A.Fio
FROM Abonent A, NachislSumma F
WHERE F.NachislSum >
      (SELECT AVG(S.NachislSum)
       FROM NachislSumma S
       WHERE F.AccountCD = S.AccountCD)
AND A.AccountCD = F.AccountCD
ORDER BY 3;
```

Результат выполнения запроса представлен на рис. 3.98.

NACHISLSUM	AVG_D	ACCOUNTCD	FIO
58,70	55,70	005488	АКСЕНОВ С.А.
56,00	55,70	005488	АКСЕНОВ С.А.
62,13	55,70	005488	АКСЕНОВ С.А.
38,32	28,32	015527	КОНЮХОВ В.С.
80,00	42,81	080047	ШУБИНА Т.П.
80,00	42,81	080047	ШУБИНА Т.П.
57,10	55,32	080270	ТИМОШКИНА Н.Г.
58,10	55,32	080270	ТИМОШКИНА Н.Г.
60,10	55,32	080270	ТИМОШКИНА Н.Г.
56,00	22,93	080613	ЛУКАШИНА Р.М.
250,00	93,49	115705	МИЩЕНКО Е.В.
250,00	93,49	115705	МИЩЕНКО Е.В.
25,30	20,30	126112	МАРКОВА В.П.
56,00	28,15	136160	ШМАКОВ С.В.
58,70	32,13	136169	ДЕНИСОВА Е.К.
80,00	48,77	443069	СТАРОДУБЦЕВ Е.В.
21,67	19,73	443690	ТУЛУПОВА М.И.

**Рис. 3.98.** Результат использования одной и той же таблицы

В этом примере производится одновременная оценка среднего значения для *всех* строк, удовлетворяющих условию поиска в предложении WHERE вложенного связанного запроса, одной и той же таблицы со значениями строки-кандидата. Выбирается первая строка-кандидат из таблицы NachislSumma и сохраняется под псевдонимом F. Выполняется вложенный запрос, просматривающий ту же самую таблицу NachislSumma с самого начала, чтобы найти все строки, где значение поля S.AccountCD - такое же, как значение F.AccountCD. Затем *по всем таким строкам в таблице* NachislSumma вложенный запрос (<скалярный\_подзапрос>) подсчитывает среднее значение поля NachislSum. Анализируется условие поиска основного запроса, чтобы проверить, превышает ли значение поля NachislSum из текущей строки-кандидата среднее значение, вычисленное вложенным запросом. Если это так, то текущая строка-кандидат выбирается для вывода. Таким образом, производятся одновременно и вычисление среднего, и отбор строк, удовлетворяющих условию.

**Примечание.** Запрос, использующий агрегатную функцию в условии поиска основного запроса (данная функция является возвращаемым элементом вложенного запроса), нельзя сформулировать с помощью техники соединения таблиц.

Рассмотрим использование соотнесенного вложенного запроса в условии поиска предложения HAVING.

Условие поиска предложения HAVING в подзапросе оценивается для *каждой группы* из внешнего запроса, а не для *каждой строки*. Следовательно, вложенный запрос будет выполняться *один раз* для каждой *группы*, выведенной из внешнего запроса, а не для каждой строки (как это было при использовании в предложении WHERE).

Например, чтобы подсчитать общие суммы начислений за услуги газоснабжения для абонентов, чьи фамилии начинаются с буквы С, можно использовать следующий соотнесенный вложенный запрос:

```
SELECT N.AccountCD, SUM(N.NachislSum)
FROM NachislSumma N
GROUP BY N.AccountCD
HAVING N.AccountCD =
(SELECT A.AccountCD FROM Abonent A
WHERE A.AccountCD = N.AccountCD AND A.Fio LIKE 'C%');
```

Результат выполнения запроса представлен на рис. 3.99.

ACCOUNTCD	SUM
136159	16,60
443069	195,10

**Рис. 3.99.** Результат использования соотнесенного подзапроса в предложении HAVING



Этот запрос выполняется следующим образом. Основной запрос один раз группирует таблицу NachislSumma по полю AccountCD. Затем для каждой группы выполняется связанный вложенный запрос, возвращая *единственное* значение поля AccountCD таблицы Abonent (т.к. поле AccountCD содержит уникальные значения).

### 3.3.2.4. Предикаты ANY и ALL

Операции сравнения можно расширить до многократного сравнения с использованием предикатов ANY и ALL. Это расширение используется при сравнении значения определенного столбца со значениями, возвращаемыми вложенным запросом (вложенный запрос представляет собой <подзапрос\_столбца>).

Предикат ANY, указанный после знака любой из операций сравнения, означает, что будет возвращено TRUE, если хотя бы для одного значения из подзапроса результат сравнения истинен.

Предикат ALL требует, чтобы результат сравнения был бы истинен для всех значений, возвращаемых подзапросом.

Рассмотрим использование предиката ANY.

Например, требуется вывести информацию об оплатах абонентами за услугу газоснабжения с кодом, равным 2, за период до 2001 года, которые превышают хотя бы одну из сумм, оплаченных за эту же услугу за 2001 год. Запрос будет выглядеть следующим образом:

```
SELECT *
FROM PaySumma
WHERE PaySum > ANY (SELECT PaySum
                     FROM PaySumma
                     WHERE PayYear=2001 AND
                          GazServiceCD=2)
AND PayYear<2001 AND GazServiceCD=2;.
```

Результат выполнения запроса представлен на рис. 3.100.

PAYFACTCD	ACCOUNTCD	GAZSERVICECD	PAYSUM	PAYDATE	PAYMONTH	PAYYEAR
2	005488	2	46,00	06.01.2001	12	2000
3	005488	2	56,00	06.05.1999	4	1999
4	115705	2	40,00	10.02.2000	1	2000
7	136160	2	56,00	12.02.1999	1	1999
9	080047	2	80,00	26.11.1998	10	1998
13	115705	2	250,00	06.10.2000	9	2000

**Рис. 3.100.** Результат использования предиката ANY

В этом примере вложенный запрос выполняется один раз, возвращая все значения поля PaySum, для которых истинно условие PayYear=2001 и GazServiceCD=2 (58.7, 250, 20, 20, 80...). Затем значения, выбранные

подзапросом, последовательно сравниваются со значением поля PaySum для каждой строки из таблицы PaySumma основного запроса. При первом обнаруженном совпадении сравнение прекращается и соответствующая строка выводится.

Условие «> ANY» равносильно утверждению "больше, чем минимальное из существующих", а условие «< ANY» - "меньше, чем максимальное из существующих". Становится очевидным, что такие условия можно записать иначе, используя агрегатные функции MIN и MAX.

Таким образом, предыдущий запрос можно переписать так:

```
SELECT *
FROM PaySumma
WHERE PaySum > (SELECT MIN (PaySum)
                FROM PaySumma
                WHERE PayYear=2001 AND
                   GazServiceCD=2)
AND PayYear<2001 AND GazServiceCD=2;
```

Результат выполнения будет совпадать с результатом, представленным на рис. 3.100.

Следует отметить, что использование сравнения «= ANY» эквивалентно использованию предиката IN.

Рассмотрим использование предиката ALL.

Например, требуется вывести информацию о ремонтных заявках абонентов, даты подачи заявок которых позднее, чем заявки любых абонентов с неисправностью с кодом, равным 2. Запрос будет выглядеть следующим образом:

```
SELECT * FROM Request
WHERE IncomingDate > ALL (SELECT IncomingDate
                          FROM Request
                          WHERE FailureCD=2);
```

Результат выполнения запроса представлен на рис. 3.101.

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
1	005488	1	1	17.12.2001	20.12.2001	1
5	080270	4	1	31.12.2001	<null>	0
9	136169	2	1	06.11.2001	08.11.2001	1
16	115705	2	3	28.12.2001	<null>	0
19	080270	4	8	17.12.2001	27.12.2001	1

**Рис. 3.101.** Результат использования предиката ALL

Если требуется вывести информацию о ремонтных заявках абонентов, даты выполнения заявок у которых позднее, чем даты выполнения заявок любых абонентов по неисправности с кодом, равным 2, то запрос будет выглядеть следующим образом:

```

SELECT * FROM Request
WHERE ExecutionDate > ALL (SELECT ExecutionDate
                           FROM Request
                           WHERE FailureCD=2);

```

В процессе выполнения данного запроса подзапросом формируется набор значений поля ExecutionDate, взятых из строк, где FailureCD=2. В результате условие поиска внешнего запроса будет выглядеть следующим образом: ExecutionDate > ALL (24.10.1998, 11.10.2001, 14.09.2001).

Результат выполнения запроса представлен на рис. 3.102.

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
1	005488	1	1	17.12.2001	20.12.2001	1
9	136169	2	1	06.11.2001	08.11.2001	1
19	080270	4	8	17.12.2001	27.12.2001	1

**Рис. 3.102.** Результат использования предиката ALL

В ТРЗ не включены строки, где поле ExecutionDate имеет значение NULL, так как условие «NULL > ALL(...)» всегда возвращает результат NULL, а выводятся только те строки, для которых условие поиска истинно.

Условие «> ALL» равносильно утверждению "больше, чем максимальное", а условие «< ALL» - "меньше, чем минимальное". Становится очевидным, что такие условия можно записать иначе, используя агрегатные функции MAX и MIN.

Таким образом, предыдущий запрос можно переписать так:

```

SELECT * FROM Request
WHERE IncomingDate > (SELECT MAX (IncomingDate)
                     FROM Request
                     WHERE FailureCD=2);

```

Результат выполнения будет таким же, как и в предыдущем примере.

Следует отметить, что использование сравнения «<> ALL» эквивалентно использованию предиката NOT IN, независимо от того, простой или связанный подзапрос используется.

Рассмотрим использование связанного подзапроса с предикатом ALL. Пусть требуется вывести названия неисправностей, по которым все ремонтные заявки подавались позднее 1 мая 2001 года. Запрос будет выглядеть следующим образом:

```

SELECT d.FailureNM
FROM Disrepair d
WHERE '01.05.2001' < ALL (SELECT r.IncomingDate
                        FROM Request r
                        WHERE d.FailureCD=r.FailureCD);

```

Результат выполнения запроса представлен на рис. 3.103.

<b>FAILURENM</b>
Засорилась водогрейная колонка
Неисправна печная горелка
Туго поворачивается пробка крана плиты

**Рис. 3.103.** Результат использования связанного подзапроса с ALL

Так как в этом примере используется связанный подзапрос, то он выполняется для каждой текущей строки из таблицы Disrepair (эта строка сохраняется во внешнем запросе под псевдонимом d). Вложенный запрос просматривает всю таблицу Request, чтобы найти строки, где значение поля r.FailureCD такое же, как значение d.FailureCD, и формирует набор значений поля IncomingDate для текущей строки-кандидата. Затем анализируется условие поиска основного запроса, чтобы проверить, меньше ли значение '01.05.2001' всех значений поля IncomingDate, полученных подзапросом. Если это так, то текущая строка-кандидат выбирается для вывода ее из основного запроса.

### 3.3.2.5. Предикат SINGULAR

Совместно с подзапросами можно использовать предикат SINGULAR, который проверяет, возвращает ли подзапрос в точности одно значение. Если возвращается NULL или более одного значения, то SINGULAR дает ложь (а NOT SINGULAR - истину). Предикат SINGULAR похож на предикат ALL, за исключением того, что он проверяет наличие одного и только одного соответствующего значения в наборе.

С предикатом SINGULAR могут использоваться как простые, так и соотнесенные подзапросы, однако часто использование простого запроса не имеет логического смысла. Следующим запросом, например, выводятся все данные об услугах газоснабжения, если оплата в размере 40 производилась только один раз:

```
SELECT *
FROM Services S
WHERE SINGULAR (SELECT PayFactCD FROM PaySumma P
                 WHERE PaySum=40);
```

Данный запрос не имеет практического смысла, так как данные об услугах и о значениях оплат никак не связаны. Поэтому с предикатом SINGULAR в основном используют соотнесенные подзапросы.

Например, следующий запрос отыскивает всех абонентов, которые имеют только одну ремонтную заявку:

```
SELECT * FROM Abonent A
WHERE SINGULAR (SELECT RequestCD FROM Request R
                 WHERE A.AccountCD=R.AccountCD);
```

Результат выполнения запроса представлен на рис. 3.104.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
015527	3	1	65	КОНЮХОВ В.С.	761699
136159	7	39	1	СВИРИНА З.А.	350003
080613	8	35	11	ЛУКАШИНА Р.М.	254417

**Рис. 3.104.** Результат использования связанного подзапроса с предикатом SINGULAR

Если использовать в предыдущем запросе предикат NOT SINGULAR, то будут выведены абоненты, у которых имеется более одной ремонтной заявки или вообще нет заявок:

```
SELECT *
FROM Abonent A
WHERE NOT SINGULAR (SELECT RequestCD FROM Request R
WHERE A.AccountCD=R.AccountCD);
```

Результат выполнения запроса представлен на рис. 3.105.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
005488	3	4	1	АКСЕНОВ С.А.	556893
115705	3	1	82	МИЩЕНКО Е.В.	769975
443690	7	5	1	ТУЛУПОВА М.И.	214833
443069	4	51	55	СТАРОДУБЦЕВ Е.В.	683014
136160	4	9	15	ШМАКОВ С.В.	982222
126112	4	7	11	МАРКОВА В.П.	683301
136169	4	7	13	ДЕНИСОВА Е.К.	680305
080047	8	39	36	ШУБИНА Т.П.	257842
080270	6	35	6	ТИМОШКИНА Н.Г.	321002

**Рис. 3.105.** Результат использования подзапроса с NOT SINGULAR

### 3.3.2.6. Предикат EXISTS

Условие EXISTS означает проверку существования. В SQL условие поиска с проверкой существования представляется следующим выражением:

[NOT] EXISTS (<табличный\_подзапрос>).

Результат условия считается истинным только тогда, когда результат выполнения <табличный\_подзапрос> является непустым множеством, т.е. когда существует какая-либо запись в таблице, указанной в предложении FROM запроса, удовлетворяющая условию поиска предложения WHERE вложенного запроса. Другими словами, EXISTS – это предикат, который возвращает значение, равное TRUE или FALSE, в зависимости от наличия вывода из вложенного запроса. Он может работать автономно в условии поиска или в комбинации с другими логическими выражениями, использующими логические операции AND, OR и NOT.

Он берет вложенный запрос как аргумент и оценивает его:

- как верный, если тот производит любой вывод;

- как неверный, если тот не делает этого.

Этим он отличается от других предикатов в условии поиска, где он не может быть неизвестным.

Например, существует возможность с помощью следующего запроса решить, извлекать ли некоторые данные из таблицы Abonent, если хотя бы у одного из абонентов имеются непогашенные заявки на ремонт газового оборудования:

```
SELECT AccountCD, Fio FROM Abonent
WHERE EXISTS
(SELECT * FROM Request WHERE Executed = 0);
```

Результат выполнения запроса представлен на рис. 3.106.

ACCOUNTCD	FIO
005488	АКСЕНОВ С.А.
115705	МИЩЕНКО Е.В.
015527	КОНЮХОВ В.С.
443690	ГУЛУПОВА М.И.
136159	СВИРИНА З.А.
443069	СТАРОДУБЦЕВ Е.В.
136160	ШМАКОВ С.В.
126112	МАРКОВА В.П.
136169	ДЕНИСОВА Е.К.
080613	ЛУКАШИНА Р.М.
080047	ШУБИНА Т.П.
080270	ТИМОШКИНА Н.Г.

**Рис. 3.106.** Результат использования условия EXISTS

В этом примере вложенный запрос выбирает все данные о непогашенных ремонтных заявках. Предикат EXISTS в условии поиска внешнего запроса "отмечает", что вложенным запросом был произведен некоторый вывод, и, поскольку предикат EXISTS был одним в условии поиска, делает условие поиска основного запроса верным. Поскольку в таблице заявок имеются (EXISTS) строки Executed = 0, то в ТРЗ представлены все строки таблицы Abonent.

В соотнесенном вложенном запросе предикат EXISTS оценивается *отдельно для каждой строки таблицы*, имя которой указано во внешнем запросе, т.е. алгоритм выполнения запроса с предикатом EXISTS и связанным подзапросом точно такой же, как и для всех запросов с соотнесенными подзапросами в условии поиска. Например, можно с помощью следующего запроса вывести коды неисправностей, которые возникали у газового оборудования нескольких абонентов:

```
SELECT DISTINCT FailureCD
FROM Request Out
WHERE EXISTS (SELECT * FROM Request Inn
              WHERE Inn.FailureCD = Out.FailureCD AND
                    Inn.AccountCD <> Out.AccountCD);
```

Результат выполнения запроса представлен на рис. 3.107.

<b>FAILURECD</b>
1
2
6
7
8
12

**Рис. 3.107.** Результат использования условия EXISTS при соотнесенном вложенном запросе

Для каждой строки-кандидата внешнего запроса внутренний запрос находит строки, совпадающие со значением в поле FailureCD и соответствующие разным абонентам (условие "AND Inn. AccountCD <> Out. AccountCD"). Если любые такие строки найдены внутренним запросом, то это означает, что имеются два разных абонента, газовое оборудование которых имело текущую неисправность (то есть неисправность в текущей строке-кандидате из внешнего запроса). Предикат EXISTS возвращает TRUE для текущей строки (т.к. результат выполнения подзапроса является непустым множеством), и номер неисправности из таблицы, указанной во внешнем запросе, будет выведен.

Если DISTINCT не указывать, то каждая из этих неисправностей будет выбираться для каждого абонента, у которого она произошла (у некоторых несколько раз).

Использование NOT EXISTS указывает на инверсию результатов запроса.

Как предикат, EXISTS можно использовать во всех случаях, когда необходимо определить, имеется ли вывод из вложенного запроса. Поэтому можно использовать предикат EXISTS и в соединении таблиц.

Например, с помощью следующего запроса можно вывести не только коды, но и названия неисправностей, которые возникали у газового оборудования нескольких абонентов:

```
SELECT DISTINCT D.FailureCD, D.FailureNM
FROM Disrepair D, Request Out
WHERE EXISTS (SELECT * FROM Request Inn
              WHERE Inn.FailureCD = Out.FailureCD
              AND Inn.AccountCD <> Out.AccountCD)
AND D.FailureCD = Out.FailureCD;
```

В этом примере внешний запрос – это соединение таблицы Disrepair с таблицей Request.

Результат выполнения запроса представлен на рис. 3.108.

FAILURECD	FAILURENM
1	Засорилась водогрейная колонка
2	Не горит АГВ
6	Плохое поступление газа на горелку плиты
7	Туго поворачивается пробка крана плиты
8	При закрытии краника горелка плиты не гаснет
12	Неизвестна

**Рис. 3.108.** Результат использования условия EXISTS при соединении

**Примечание.** Предикат EXISTS нельзя использовать в случае, если вложенный запрос возвращает значение агрегатной функции.

### 3.3.3. Объединение результатов нескольких запросов

При получении данных из таблиц БД необходимость в объединении результатов двух или более запросов в одну таблицу реализуется с помощью предложения UNION. Предложение UNION объединяет вывод двух или более запросов в единый набор строк и столбцов и имеет вид:

```
Запрос_X UNION [{DISTINCT | ALL}]
Запрос_Y UNION [{DISTINCT | ALL}]
Запрос_Z...
```

Чтобы таблицы результатов нескольких запросов можно было объединять с помощью предложения UNION, они должны соответствовать *следующим требованиям*:

- содержать одинаковое число столбцов;
- тип данных каждого столбца любой таблицы должен совпадать с типом данных соответствующего столбца любой другой таблицы;
- ни одна из таблиц промежуточного запроса не может быть отсортирована с помощью предложения ORDER BY;
- разрешается использовать в списке возвращаемых элементов только имена столбцов или указывать все столбцы (SELECT \*) и запрещается использовать выражения.

Объединение таблиц с помощью предложения UNION отличается от вложенных запросов и соединений таблиц тем, что в нем ни один из двух (или больше) запросов не управляет другим запросом. Все запросы выполняются независимо друг от друга, а уже их *вывод объединяется*. Например, необходимо вывести как единое целое всех абонентов и исполнителей ремонтных заявок, фамилии которых начинаются на букву «Ш». Для этого можно использовать следующий запрос:

```
SELECT Fio AS AbonentFio FROM Abonent
WHERE Fio LIKE 'Ш%'
UNION
SELECT Fio FROM Executor WHERE Fio LIKE 'Ш%';
```

Результат объединения представлен на рис. 3.109.



<b>ABONENTFIO</b>
ШКОЛЬНИКОВ С.М.
ШЛЮКОВ М.К.
ШМАКОВ С.В.
ШУБИН В.Г.
ШУБИНА Т.П.

**Рис. 3.109.** Результат объединения двух запросов

**Примечания.**

1. Имена атрибутов в ТРЗ берутся как имена возвращаемых элементов в первом запросе (или псевдоним столбца, как в предыдущем примере).
2. Только последний запрос заканчивается точкой с запятой. Отсутствие точки с запятой после первого запроса дает понять SQL, что имеется еще один или более запросов.

Объединение таблиц с помощью предложения UNION DISTINCT используется как синоним просто UNION для автоматического исключения дубликатов строк из вывода. Однако в соответствии со стандартом исключение дубликатов строк из вывода является режимом по умолчанию, поэтому слово DISTINCT использовать необязательно.

Чтобы включить все строки в вывод запроса, следует указать UNION ALL. Допустим, если бы был не только исполнитель с именем Школьников С.М., но и абонент с таким же именем и вместо UNION использовался бы UNION ALL, то тогда строка с именем Школьников С.М. была бы выведена два раза.

Предложения UNION и UNION ALL могут быть скомбинированы, чтобы удалять одни дубликаты, не удаляя других. Объединение запросов:

(Запрос\_X UNION ALL Запрос\_Y) UNION Запрос\_Z;

не обязательно даст те же результаты, что объединение запросов:

Запрос\_X UNION ALL (Запрос\_Y UNION Запрос\_Z);,

т.к. дублирующиеся строки удалятся при использовании UNION без ALL.

Результаты выполнения промежуточных запросов, участвующих в объединении, упорядочивать запрещено, однако результирующий набор можно *отсортировать*, но только указывая порядковые номера для определения порядка столбцов.

Например, объединить в одну таблицу информацию об услугах газоснабжения и неисправностях газового оборудования, а результат отсортировать по наименованию неисправности в обратном алфавитном порядке можно с помощью следующего запроса:

```
SELECT FailureCD, FailureNM FROM Disrepair
UNION ALL
SELECT GazServiceCD, GazServiceNM FROM Services
ORDER BY 2 DESC;
```

Результат объединения представлен на рис. 3.110.

FAILURECD	FAILURENM
7	Туго поворачивается пробка крана плиты
3	Течет из водогрейной колонки
8	При закрытии краника горелка плиты не гаснет
6	Плохое поступление газа на горелку плиты
4	Неисправна печная горелка
5	Неисправен газовый счетчик
12	Неизвестна
2	Не горит АГВ
2	Заявочный ремонт ГО
1	Засорилась водогрейная колонка
1	Доставка газа

**Рис. 3.110.** Результат объединения двух запросов с сортировкой

В объединяемых запросах, если требуется, можно использовать *одну и ту же таблицу*.

Пусть, например, требуется вывести начисленные суммы за 2001 год, уменьшенные на 5%, если сумма меньше 30, на 10%, если сумма от 30 до 100, и на 20%, если сумма больше 100. Вывести также процент уменьшения, код начисления, прежнюю и новую начисленные суммы. Запрос будет выглядеть следующим образом:

```

SELECT AccountCD,'Снижение - 5%', NachisIFactCD,
        NachisSum AS Old_Sum,
        NachisSum*0.95 AS New_Sum
FROM NachisSumma
WHERE NachisSum < 30 AND NachisYear = 2001
UNION
SELECT AccountCD,'Снижение - 10%', NachisIFactCD,
        NachisSum, NachisSum*0.90
FROM NachisSumma
WHERE (NachisSum between 30 and 100)
AND NachisYear=2001
UNION
SELECT AccountCD,'Снижение - 20%', NachisIFactCD,
        NachisSum, NachisSum*0.80
FROM NachisSumma WHERE NachisSum > 100 AND NachisYear =
2001;

```

Результат объединения представлен на рис. 3.111.

ACCOUNTCD	F_1	NACHISLFACTCD	OLD_SUM	NEW_SUM
005488	Снижение - 10%	19	58,70	52,8300
080047	Снижение - 10%	8	80,00	72,0000
080047	Снижение - 10%	35	32,56	29,3040
080270	Снижение - 10%	9	46,00	41,4000

080270	Снижение - 10%	43	60,10	54,0900
080613	Снижение - 10%	10	56,00	50,4000
115705	Снижение - 10%	12	58,70	52,8300
115705	Снижение - 10%	31	37,80	34,0200
115705	Снижение - 20%	5	250,00	200,0000
126112	Снижение - 5%	46	25,30	24,0350
136160	Снижение - 5%	13	20,00	19,0000
136169	Снижение - 10%	16	58,70	52,8300
136169	Снижение - 5%	15	20,00	19,0000
136169	Снижение - 5%	39	28,32	26,9040
443069	Снижение - 10%	17	80,00	72,0000
443069	Снижение - 10%	18	38,50	34,6500
443069	Снижение - 10%	47	38,32	34,4880

**Рис. 3.111.** Результат использования одной и той же таблицы при объединении запросов

Следует отметить, что объединения результатов нескольких запросов в один результирующий набор разрешены и *внутри подзапросов*.

Например, требуется вывести все данные об оплаченных суммах за услугу газоснабжения с кодом, равным 1, по абонентам, фамилия которых начинается с буквы М, и по абонентам, которые подавали заявки с неисправностью газового оборудования с кодом, равным 1. Запрос будет выглядеть следующим образом:

```
SELECT * FROM PaySumma
WHERE AccountCD IN
(SELECT AccountCD FROM Abonent WHERE Fio Like 'M%' UNION
SELECT R.AccountCD FROM Request R WHERE FailureCD=1)
AND GazServiceCD=1
ORDER BY AccountCD;
```

Результат объединения представлен на рис. 3.112.

PAYFACTCD	ACCOUNTCD	GAZSERVICECD	PAYSUM	PAYDATE	PAYMONTH	PAYYEAR
29	005488	1	62,13	03.05.2000	4	2000
27	080270	1	57,10	05.03.1998	2	1998
37	080270	1	58,10	07.01.2001	12	2000
42	080270	1	60,10	07.06.2001	5	2001
24	115705	1	37,15	04.11.1999	10	1999
30	115705	1	37,80	12.07.2001	5	2001
36	115705	1	37,15	23.12.1999	11	1999
48	115705	1	37,15	10.08.1999	6	1999
33	126112	1	15,30	08.09.2000	8	2000
45	126112	1	25,30	10.09.2001	8	2001
26	136169	1	25,32	03.02.1999	1	1999
38	136169	1	28,32	08.02.2001	1	2001
43	136169	1	28,32	05.03.1998	2	1998

**Рис. 3.112.** Результат объединения запросов внутри подзапроса

Из результата следует, что информация об оплатах выведена по абонентам, которые подавали заявки по неисправности с кодом, равным 1 (лицевые счета '005488', '080270', '115705', '136169'), и по абонентам, фамилия которых начинается с буквы М (лицевые счета '115705', '126112'). Так как для

объединения использовали предложение UNION, то исключены повторяющиеся строки по абоненту с номером лицевого счета, равным '115705', информация о котором возвращается в каждом из объединяемых с помощью UNION запросов.

## Контрольные вопросы

1. Какие предложения включает в себя запрос SELECT? В какой последовательности выполняется обработка элементов запроса SELECT?
2. Что может представлять собой возвращаемый элемент в запросе SELECT?
3. Какие функции выполняют конструкции FIRST...SKIP и ROWS?
4. Каким образом может быть задано условие поиска применительно к однотабличным запросам?
5. Какие виды функций используются в СУБД Firebird? Опишите особенности синтаксиса и работы нескольких функций каждого вида.
6. Как отсортировать результаты запроса на выборку данных?
7. Как сформировать запрос с группировкой? Какие ограничения накладываются на такие запросы?
8. Каким образом используется предложение HAVING в запросе SELECT? В чем состоит отличие в работе предложений HAVING и WHERE?
9. Какие существуют варианты запросов на выборку к множеству таблиц?
10. Что такое неявное и явное соединение? Какие виды явного соединения существуют? В каком случае неявное и явное соединения взаимозаменяемы?
11. Что такое вложенный запрос? В каких предложениях запроса SELECT могут использоваться вложенные запросы?
12. Как создать рекурсивный запрос на языке SQL?
13. Что такое соотнесенный вложенный запрос? Чем отличается работа соотнесенного вложенного запроса от работы простого подзапроса?
14. Каким образом используются предикаты ANY и ALL с подзапросами? Как с помощью агрегатных функций можно заменить выражения с предикатами ANY и ALL?
15. Каким образом используются предикаты SINGULAR и EXISTS с подзапросами?
16. Как выполняется объединение результатов нескольких запросов?

## 4. Язык определения данных

Для создания и изменения структуры БД предназначены SQL-запросы, называемые языком определения данных, или DDL (Data Definition Language). С помощью DDL-запросов можно выполнить следующее:

- определить структуру нового домена и создать его;
- определить структуру новой таблицы и создать ее;
- изменить определение существующей таблицы;
- определить виртуальную таблицу (представление, курсор);

– создать индексы для ускорения доступа к таблицам.

Ядро языка определения данных в Firebird образуют следующие четыре SQL-запроса:

- CREATE (создать), позволяющий определить и создать объект БД;
- DROP (удалить), служащий для удаления существующего объекта БД;
- ALTER (изменить), с помощью которого можно изменить определение объекта БД;
- RECREATE (заново создать), с помощью которого можно заново создать объект со старым именем.

Пустая БД в Firebird представляет собой файл, который не содержит таблиц пользователя, но содержит пустые системные таблицы. Физическая структура пустой БД (файл) создается сервером Firebird при выполнении скрипта, содержащего команду CREATE DATABASE, или в интерактивном режиме утилиты IBExpert. Для *подключения* к существующей БД можно воспользоваться соответствующим пунктом меню утилиты IBExpert или использовать скрипт, содержащий команду CONNECT.

Таким образом, чтобы создать БД в Firebird необходимо:

1. Создать пустой файл БД и выполнить команду соединения с базой.
2. Создать необходимые домены.
3. Создать пользовательские таблицы.
4. Заполнить данными созданные таблицы пользователя.

Команды CREATE DATABASE и CONNECT будут рассмотрены позднее при изучении SQL-скриптов.

В настоящей главе рассматриваются задачи определения доменов, создания, изменения и удаления постоянных и временных пользовательских таблиц. Дается понятие индексов и описывается их роль в повышении эффективности выполнения операторов SQL. Здесь же рассмотрены запросы создания и изменения индексов. Все это относится к самим таблицам, а не к данным, которые в них содержатся.

Также в данной главе приводится описание правил создания представлений различных видов, используемых для ограничения доступа отдельных пользователей к различным данным.

## 4.1. Домены

Если в таблицах БД имеются столбцы, обладающие одними и теми же характеристиками, то можно предварительно определить с помощью домена тип данных и поведение таких столбцов, а затем поставить в соответствие каждому из однотипных столбцов имя соответствующего домена.

*Доменом* называется именованное множество скалярных значений одного типа. Например, домен TMonth (месяц) в учебной базе данных – это множество всех возможных номеров месяцев (от 1 до 12). Цель доменов в SQL – обеспечить возможность один раз определить элементарную спецификацию типа данных, а затем использовать ее одновременно для нескольких столбцов в нескольких базовых таблицах.

То есть домен – это тип данных (как это понимается в современных языках программирования). Например, в языке программирования Pascal допустимы следующие выражения:

```
type TDay = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);  
var Today : TDay;
```

Здесь имеется определенный пользователем тип TDay (имеющий в точности семь допустимых значений) и переменная Today, принадлежащая этому типу данных (а значит, и ограниченная этими семью значениями). Эта ситуация аналогична ситуации в реляционной БД, когда имеются домен, названный TDay, и атрибут, названный Today.

Основной особенностью доменов является то, что *домены ограничивают сравнения*. Это значит, что сравнение атрибутов, определенных на основе одного домена, имеет смысл, а сравнение атрибутов, определенных на основе различных доменов, бессмысленно. Домены имеют концептуальную природу. Они должны быть определены в рамках конкретной БД. Тогда каждое определение атрибута (например, столбца определенной таблицы) должно включать ссылку на соответствующий домен. Таким образом, системе будет известно, какие атрибуты можно сравнивать, а какие – нет.

Синтаксис запроса определения домена имеет следующий вид:

```
CREATE DOMAIN имя_домена [AS] <тип_данных>  
[DEFAULT { литерал | NULL | USER}]  
[NOT NULL] [CHECK ( <ограничение_домена>)];
```

где

имя\_домена – имя создаваемого домена;

<тип\_данных> – тип данных, представленный в табл. 2.5;

DEFAULT– ключевое слово, определяющее значение по умолчанию, применяемое к каждому столбцу как:

- литерал – константа строкового, числового типа или типа дата/время (соответствует типу данных домена);
- NULL – значение NULL;
- USER – контекстная переменная, возвращающая имя пользователя, подключенного к БД.

Следующие предложения задают набор ограничений целостности к каждому столбцу, определенному на этом домене:

- NOT NULL – чтобы столбец не имел NULL-значения. Этот атрибут используется при определении домена, если требуется, чтобы все столбцы, создаваемые на основе этого домена, имели непустое значение. Следует помнить, что переопределить атрибут NOT NULL, заданный для домена, нельзя. Часто неизвестно, действительно ли потребуется, чтобы все столбцы, определяемые на домене, имели NOT NULL значения. Также часто требуется определять внешний ключ на домене без условия NOT NULL. Поэтому предпочтительнее добавлять атрибут NOT NULL при определении столбцов (будет рассмотрено далее);

- CHECK – определяет список ограничений на значение (VALUE) в соответствующем столбце. Ограничение домена фактически повторяет синтаксис условия поиска в предложении WHERE для однотабличных запросов (используются простое сравнение, проверка на принадлежность диапазону, на членство во множестве и т.д.) с той лишь разницей, что вместо точного указания проверяемого значения используется слово VALUE.

Ограничение домена может быть одним из следующих:

```
<ограничение_домена> ::= [NOT] <ограничение_домена1>
                        [[AND|OR][NOT] <ограничение_домена2>]....
```

где

```
<ограничение_домена> ::=
{VALUE <операция_сравнения> <значение>
| VALUE [NOT] BETWEEN <значение1> AND <значение2>
| VALUE [NOT] LIKE 'шаблон' [ESCAPE 'символ_пропуска' ]
| VALUE [NOT] CONTAINING <значение>
| VALUE [NOT] STARTING [WITH] <значение>
| VALUE [NOT] IN (<значение1> [ , <значение2> ...])
| VALUE IS [NOT] NULL
| VALUE IS [NOT] DISTINCT FROM <значение>;
```

```
<значение> ::= { столбец | константа | <выражение> | функция}.
```

Ключевое слово VALUE используется как обозначение значения, которое будет помещаться в столбец таблицы, имеющий тип соответствующего домена.

Например, для определения домена с именем Telephone, описывающего номер телефона абонента (по умолчанию '999999' и не может быть значение '100000') и имеющего тип VARCHAR(10), следует применить следующий запрос:

```
CREATE DOMAIN Telephone AS VARCHAR(10)
DEFAULT '999999'
CHECK (VALUE <> '100000');
```

После определения домена его имя используется для определения типа соответствующих столбцов таблиц.

**Примечание.** Если в таблице присутствует один столбец и он имеет значение по умолчанию, то его не удастся использовать, так как требуется при вставке в таблицу указать явно хотя бы один столбец.

В учебной базе данных определены пять доменов, запросы определения которых выглядят следующим образом:

```
CREATE DOMAIN Boolean AS SMALLINT
CHECK (VALUE IN (0, 1));
CREATE DOMAIN Money AS NUMERIC(15,2);
CREATE DOMAIN PKField AS INTEGER;
CREATE DOMAIN TMonth AS SMALLINT
CHECK (VALUE BETWEEN 1 AND 12);
```

```
CREATE DOMAIN TYear AS SMALLINT  
CHECK (VALUE BETWEEN 1990 AND 2100);
```

Просмотреть список доменов, определенных в БД, и структуру каждого из них можно в IBExpert с помощью инспектора объектов.

Определение существующего домена можно *изменить* с помощью запроса ALTER DOMAIN. Этот запрос позволяет:

- удалить существующее и определить новое значение по умолчанию (заменяя при этом старое значение, если оно было указано);
- удалить существующее и ввести новое ограничение целостности.

Синтаксис запроса ALTER DOMAIN имеет следующий формат:

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

где

SET DEFAULT – указывает для существующего домена значение по умолчанию;

DROP DEFAULT - для существующего домена удаляет значение по умолчанию;

ADD [CONSTRAINT] CHECK - добавляет ограничение для существующего домена;

DROP CONSTRAINT – удаляет CHECK ограничение из определения домена.

Например, созданный домен Telephone можно изменить, удалив ограничение, следующим запросом:

```
ALTER DOMAIN Telephone DROP CONSTRAINT;
```

или установив другое значение по умолчанию следующим запросом:

```
ALTER DOMAIN Telephone SET DEFAULT '111111';
```

Существующий домен можно *удалить* с помощью запроса DROP DOMAIN, имеющего следующий синтаксис:

```
DROP DOMAIN имя_домена;
```

**Примечание.** Домен не будет удален, если на него имеются какие-либо ссылки, т.е. существуют таблицы со столбцами, определенными на этом домене.

Например, если попытаться на учебной БД удалить домен BOOLEAN:

```
DROP DOMAIN BOOLEAN;;
```

то будет выдано сообщение об ошибке следующего вида: "column BOOLEAN is used in table REQUEST (local name EXECUTED) and cannot be dropped" [столбец BOOLEAN используется в таблице REQUEST (локальное имя EXECUTED) и не может быть удален].



## 4.2. Создание, изменение и удаление базовых таблиц БД

### 4.2.1. Создание таблицы

В реляционной БД наиболее важным элементом ее структуры является таблица. Перед тем как перейти к созданию таблиц, необходимо выполнить проектирование базы данных и нормализацию таблиц.

Кроме того, приступая к созданию таблицы, необходимо иметь ответы на следующие вопросы.

Как будет называться таблица?

Как будут называться столбцы (поля) таблицы?

Какие типы данных будут закреплены за каждым столбцом?

Какие столбцы таблицы требуют обязательного ввода?

Из каких столбцов будет состоять первичный ключ?

Для создания базовой таблицы, ее столбцов и ограничений, налагаемых на каждый столбец, используется запрос CREATE TABLE, который имеет следующий формат:

```
CREATE TABLE базовая_таблица (<определение_столбца1>  
[, <определение_столбца2>.....]  
[,<тип_ограничения>.....]);
```

Таким образом, элементы в круглых скобках после имени таблицы могут представлять собой как определение столбца, так и ограничения базовой таблицы.

Конструкция <определение\_столбца> имеет следующий вид:

```
<определение_столбца> ::= столбец  
{ <тип_данных> | COMPUTED [BY] (<выражение>) | имя_домена }  
[NOT NULL]  
[DEFAULT { литерал | NULL | USER}]  
[<ограничение_столбца>].
```

Все столбцы таблицы должны иметь уникальные имена. Тип данных столбца может задаваться непосредственно указанием имени типа или указанием имени домена. С точки зрения теории БД использование доменов при определении типа столбца является необходимым. При этом домены должны создаваться до создания таблиц.

Конструкция

COMPUTED [BY] (<выражение>)

определяет, что значение столбца вычисляется во время выполнения запроса в соответствии с указанным выражением. При этом выражение должно возвращать одно значение, а столбцы таблицы, указанные в выражении, должны существовать до их использования.

Например, если требуется поместить в столбец `summa` значение суммы столбцов `first` и `second`, описание типа данных столбца `summa` должно выглядеть так:

```
summa COMPUTED BY (first + second).
```

В результате тип данных столбца `summa` будет автоматически приведен к типу данных столбцов `first` и `second` по правилу преобразования типов.

При конкатенации строк, содержащих имя (`name`) и отчество (`second_name`), для получения полного имени (`full_name`) описание типа данных столбца `full_name`, содержащего в качестве разделителя запятую, может быть следующим:

```
full_name COMPUTED BY (name || ',' || second_name).
```

Столбец `full_name` будет иметь строковый тип данных с длиной, равной сумме длин строк `second_name` и `name`.

Очевидно, что определенные таким образом столбцы `summa` и `full_name` будут доступны только для чтения (`read-only column`), т.е. в эти столбцы нельзя добавлять значения, а следовательно, они не должны упоминаться в списке столбцов запросов `INSERT` и `UPDATE`.

Предложение `NOT NULL` устанавливает ограничение на непустое значение столбца (условие обязательности данных). Так, например, поле первичного ключа не может содержать значений `NULL`, следовательно, при его описании должно быть использовано ограничение `NOT NULL`.

Предложение `DEFAULT` определяет значение столбца по умолчанию, т.е. то значение, которое будет вставляться в таблицу при добавлении новой строки. Если таблица содержит один столбец, то ему не может быть назначено значение по умолчанию.

Например, необходимо создать таблицу `Days`, состоящую из пяти столбцов: `number` типа `INTEGER`, `dat` типа `DATE`, `event` типа `VARCHAR(20)`, `usr` типа `CHAR(10)` и `tel`, определенного на созданном домене `Telephone`. При этом столбец `number` не должен содержать `NULL`-значений, столбец `dat` по умолчанию должен содержать 1 сентября 1996 года, столбец `event` по умолчанию должен содержать `NULL`-значения, столбец `usr` по умолчанию должен содержать имя пользователя, подключенного к БД, а столбец `tel` должен содержать значение по умолчанию из домена `Telephone`. Запрос для создания таблицы `Days` примет следующий вид:

```
CREATE TABLE Days (number INTEGER NOT NULL,  
dat DATE DEFAULT '09/01/1996',  
event VARCHAR(20) DEFAULT NULL,  
usr CHAR(10) DEFAULT USER,  
tel Telephone);
```

Если определение столбца основано на домене, оно может включать новое значение по умолчанию и/или дополнительные ограничения, которые перекрывают значения, заданные при определении домена. Например, можно добавить ограничение `NOT NULL` для столбца, если домен его еще не содержит. Однако домен, который был определен как `NOT NULL`, не может быть переопределен на уровне столбца как допускающий `NULL` значение.

Для поддержания целостности БД на таблицы накладываются ограничения (CONSTRAINT). Под ограничением понимается условие, которое должно выполняться при хранении, обновлении и добавлении данных в таблицу БД. Различают ограничения, накладываемые:

- на определенный столбец;
- на всю таблицу.

#### 4.2.2. Определение ограничений столбца

При *ограничении столбца* соответствующее ограничение объявляется индивидуально для каждого столбца непосредственно после определения имени и типа столбца. При этом используется конструкция <ограничение\_столбца>.

Определение ограничения таблицы осуществляется после описания всех столбцов, и при этом используется конструкция <тип\_ограничения>, имеющая следующий формат:

```
<ограничение_столбца>::=  
  [CONSTRAINT имя_ограничения]  
  { UNIQUE  
  | PRIMARY KEY  
  | CHECK (<условие_проверки>)  
  | REFERENCES родительская_таблица [(столбец)]  
  [ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]  
  [ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}}].
```

Рекомендуется ограничениям давать имена, т.к. при использовании запроса ALTER TABLE удалить ограничение можно только по его имени.

При создании таблицы имя ограничения задается произвольно, но должно быть уникальным для данной таблицы и, желательно, отражать смысл ограничения.

Если «CONSTRAINT имя\_ограничения» отсутствует, то СУБД присваивает ограничению свое системное имя. Имя ограничения можно посмотреть в IBExpert на закладке «Ограничения» для каждого столбца конкретной таблицы.

Параметр UNIQUE определяет, что данный столбец должен иметь уникальные значения, т.е. при изменении данных в таблице (добавлении или обновлении строк) автоматически будет осуществляться проверка, что подобного значения в столбце нет.

Предложение PRIMARY KEY определяет столбец в качестве первичного ключа. Следует обратить внимание на то, что столбец таблицы, определенный как первичный ключ, должен обязательно иметь ограничение NOT NULL.

Рассмотрим запрос CREATE TABLE с ограничением определенного столбца. Например, чтобы создать таблицу Abonent, определив поле AccountCD в качестве первичного ключа, необходимо использовать следующий запрос:

```
CREATE TABLE Abonent
(AccountCD VARCHAR(6) NOT NULL
 CONSTRAINT xpka PRIMARY KEY StreetCD INTEGER,
 HouseNo SMALLINT,
 FlatNo SMALLINT,
 Fio VARCHAR(20),
 Phone VARCHAR(15));
```

Таким образом, при создании таблицы Abonent определяется ограничение с именем xpka, указывающее в качестве первичного ключа таблицы столбец AccountCD.

Для каждого столбца можно назначить условие проверки указанием в ограничении столбца предложения

CHECK (<условие\_проверки>).

При этом каждый раз, когда в такой столбец будут добавляться новые или обновляться существующие данные, автоматически будет происходить их проверка в соответствии с <условием проверки>, которое имеет следующий формат:

```
<условие_проверки> ::= [NOT] <условие_проверки1>
                        [[AND|OR][NOT] <условие_проверки2>]...,
```

где

```
<условие_проверки> ::=
{<значение> <операция_сравнения> { <значение1> | (<скалярный_подзапрос>)
                                | {ANY| ALL} (<подзапрос_столбца>)}
| <значение> [NOT] BETWEEN <значение1> AND <значение2>
| <значение> [NOT] LIKE 'шаблон' [ESCAPE 'символ пропуска']
| <значение> [NOT] CONTAINING <значение1>
| <значение> [NOT] STARTING [WITH] <значение1>
| <значение> [NOT] IN (<значение1> [ , <значение2> ...] | <подзапрос_столбца>)
| <значение> IS [NOT] NULL
| <значение> IS [NOT] DISTINCT FROM <значение1>
| EXISTS (<табличный_подзапрос>)
| SINGULAR (<табличный_подзапрос>)},
```

где

```
<значение> ::= { столбец | константа | <выражение> | функция};
```

```
<операция_сравнения> ::= {= | < | > | <= | >= | <> };
```

<табличный\_подзапрос> ::= запрос select, возвращающий набор строк и столбцов;

<подзапрос\_столбца> ::= запрос select, возвращающий значения одного столбца, но, возможно, в нескольких строках;

<скалярный\_подзапрос> ::= запрос select, возвращающий значение одного столбца в одной строке.

По существу, <условие\_проверки> – это не что иное, как условие поиска предложения WHERE при использовании вложенных запросов.

Следует отметить, что если столбец таблицы определен на домене, имеющем ограничение CHECK, то это ограничение не может быть переопределено в определении столбца, хотя столбец может расширить использование ограничения CHECK домена, добавив свои собственные условия.

Например, необходимо создать таблицу NachislSumma, определив поле NachislFactCd как первичный ключ на домене PKField. При этом начисленная сумма (поле NachislSum на домене Money) не должна быть меньше 5000, значение поля NachislYear, определяемого на домене TYear с ограничением (VALUE BETWEEN 1990 AND 2100), должно отличаться от 1995.

Запрос на создание такой таблицы будет выглядеть следующим образом:

```
CREATE TABLE NachislSumma
(NachislFactCD PKField NOT NULL PRIMARY KEY,
AccountCD VARCHAR(30) NOT NULL,
GazServiceCD PKField NOT NULL,
NachislSum Money CHECK (NachislSum >= 5000),
NachislMonth TMonth,
NachislYear TYear CHECK (NachislYear IS DISTINCT FROM
1995));
```

Если значения, помещаемые в таблицу NachislSumma, не будут удовлетворять указанным условиям проверки, то возникнет ошибка с SQLCODE=-297.

Предложение REFERENCES, указанное в качестве ограничения столбца, задает, что данный столбец таблицы ссылается на родительскую таблицу и является внешним ключом. Если после имени таблицы, на которую ссылается данный внешний ключ, не указаны имена столбцов, то подразумевается, что данный столбец ссылается на первичные ключи.

Следует обратить внимание на то, что внешний ключ может ссылаться на первичный ключ той же самой таблицы, т.е. может быть реализовано рекурсивное отношение.

Предложения ON DELETE и ON UPDATE используются вместе с REFERENCES при определении внешнего ключа и предназначены для описания типа изменения внешнего ключа при изменении соответствующего ему значения первичного ключа. Т.е. для столбца внешнего ключа таблицы-потомка задаются действия, автоматически выполняемые при удалении или обновлении поля первичного ключа в таблице-родителе, на который ссылается данный внешний ключ.

Для указания действий, которые должны выполняться над полем внешнего ключа при удалении и обновлении данных поля первичного ключа, используются следующие параметры:

– NO ACTION (используется по умолчанию) означает, что удаление или обновление первичного ключа родительской таблицы не изменяет ссылающийся внешний ключ, вследствие чего попытка операции над родительской таблицей может закончиться неудачей;

- CASCADE для ON DELETE удаляет строки, содержащие значение ссылающегося внешнего ключа, а для ON UPDATE обновляет ссылающийся внешний ключ новым значением первичного ключа;
- SET DEFAULT устанавливает значение ссылающегося внешнего ключа в заданное для него значение по умолчанию;
- SET NULL устанавливает значение ссылающегося внешнего ключа в NULL.

Предложения ON DELETE и ON UPDATE могут использоваться одновременно (т.е. для столбца в одном ограничении могут быть указаны действия, которые необходимо выполнить при удалении, а также действия, которые необходимо выполнить при обновлении).

Например, необходимо создать таблицу Request, определив поле AccountCD в качестве внешнего ключа, ссылающегося на первичный ключ таблицы Abonent. Необходимо также, чтобы при удалении поля первичного ключа в таблице Abonent удалялись строки с соответствующим значением внешнего ключа в таблице Request. При обновлении первичного ключа в таблице Abonent должно происходить обновление соответствующего внешнего ключа в таблице Request. Следующий запрос создает требуемую таблицу:

```
CREATE TABLE Request
(RequestCD INTEGER NOT NULL PRIMARY KEY,
 AccountCD VARCHAR(6) REFERENCES Abonent(AccountCD)
 ON DELETE CASCADE ON UPDATE CASCADE,
 ExecutorCD INTEGER,
 FailureCD INTEGER,
 IncomingDate DATE,
 ExecutionDate DATE,
 Executed SMALLINT);
```

#### 4.2.3. Определение ограничений на таблицу

При определении ограничений, накладываемых на всю таблицу, используется следующая синтаксическая конструкция:

```
<тип_ограничения>::=
[CONSTRAINT имя_ограничения]
{{ PRIMARY KEY | UNIQUE } (<список_столбцов>)
 | FOREIGN KEY (<список_столбцов>)
 REFERENCES родительская_таблица [(столбец1 [, столбец2 ...])]
 [ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
 [ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
 | CHECK (<условие_проверки> )
 [USING [ASC[ENDING] | DESC[ENDING]] INDEX имя_индекса].
```

Существует *три вида ограничений* базовой таблицы:

- ограничение первичного ключа;

- ссылочное ограничение (ограничение внешнего ключа);
- ограничение "проверочного условия".

При задании ограничения таблицы можно определить первичный ключ (как простой, так и составной). Если первичный ключ не является составным, то его можно определить при задании ограничения столбца (что было рассмотрено выше), но составной первичный ключ можно определить только с помощью ограничения таблицы. Для этого поля таблицы, являющиеся составным первичным ключом, перечисляются в скобках через запятую в предложении PRIMARY KEY.

Аналогично при задании ограничения таблицы можно определить как простой, так и составной внешний ключ, причем если простой ключ уже был определен в ограничении столбца, то определять его снова не требуется. Составной внешний ключ можно определить только с помощью ограничения таблицы, и для этого используется предложение FOREIGN KEY со списком столбцов и предложением REFERENCES с именами содержащих их таблиц, на которые ссылается данный внешний ключ.

Использование параметров NO ACTION, CASCADE, SET DEFAULT и SET NULL в предложениях ON DELETE и ON UPDATE аналогично использованию их при определении ограничения столбца (было рассмотрено выше).

Таким образом, если определяется простой ключ (первичный или внешний), то его можно задать как в ограничении столбца, так и в ограничении таблицы - результат будет одинаков. Однако если ключ составной, то его можно определить только в ограничении на таблицу.

Например, чтобы при создании таблицы NachislSumma определить составной первичный ключ, состоящий из полей NachislFactCD и NachislYear, и внешние ключи AccountCD и GazServiceCD, необходимо применить следующий запрос:

```
CREATE TABLE NachislSumma
(NachislFactCD INTEGER NOT NULL,
 NachislSum NUMERIC(15,2),
 NachislYear SMALLINT NOT NULL,
 NachislMonth SMALLINT,
 AccountCD VARCHAR(6) NOT NULL,
 GazServiceCD INTEGER NOT NULL,
 PRIMARY KEY (NachislFactCD, NachislYear),
 FOREIGN KEY (AccountCd) REFERENCES Abonent(AccountCD),
 FOREIGN KEY (GazServiceCD) REFERENCES
 Services(GazServiceCD));
```

При этом в качестве действия на изменение первичных ключей таблиц Abonent и Services будет определено действие NO ACTION, используемое по умолчанию.

При использовании предложения UNIQUE накладывается условие уникальности на значения столбцов таблицы, перечисленных в скобках через запятую. В столбцах с уникальными ограничениями можно использовать NULL значения. Следует учесть, что если в столбцы, на значения которых наложено

условие уникальности, пытаться ввести только NULL значения, то они не будут считаться одинаковыми и команда будет выполнена успешно.

Создадим, например, таблицу Phone\_Sprav с четырьмя столбцами: столбец number типа INTEGER, столбец Fio типа VARCHAR(25), столбец Home\_Phone типа VARCHAR(6) и столбец Mobil\_Phone типа VARCHAR(11). При этом столбец number определим в качестве первичного ключа, а в качестве ограничения на таблицу определим уникальность столбцов Home\_Phone и Mobil\_Phone. Запрос CREATE TABLE для создания таблицы Phone\_Sprav примет следующий вид:

```
CREATE TABLE Phone_Sprav (number INTEGER PRIMARY KEY,  
Fio VARCHAR(25),  
Home_Phone VARCHAR(6),  
Mobil_Phone VARCHAR(11),  
UNIQUE (Home_Phone, Mobil_Phone));
```

Если попытаться добавить в таблицу строки (позже будет рассмотрено, как это можно сделать), в которых значения полей number и Fio различны, а поля Home\_Phone и Mobil\_Phone равны NULL, то вставка этих данных пройдет успешно, так как считается, что каждое (NULL, NULL) в уникальных столбцах (Home\_Phone, Mobil\_Phone) отличается от любого другого (NULL, NULL). Однако попытка вставить строки с одинаковыми значениями в одном из уникальных столбцов (Home\_Phone или Mobil\_Phone) и с NULL значениями в другом не удастся.

Таким образом, если хотя бы для одного столбца из списка, на который наложено условие уникальности, повторяется значение, отличное от NULL, то запрос не будет выполнен.

Ограничение

CHECK (<условие\_проверки>)

может использоваться не только при определении отдельного столбца, что было рассмотрено выше, но и в качестве табличного ограничения. Это полезно в тех случаях, когда условие необходимо задать на значениях нескольких столбцов. Синтаксис <условия\_проверки> фактически повторяет рассмотренный выше для определения ограничения столбца, но могут использоваться имена нескольких различных столбцов таблицы, на которую накладывается ограничение.

Предположим, что поступившая ремонтная заявка должна выполняться не позднее, чем через неделю после поступления (т.е. дата выполнения должна быть в диапазоне от дня поступления до даты на 7 дней позднее). Запрос на создание таблицы Request с учетом этого ограничения можно представить в следующем виде:

```
CREATE TABLE Request  
(RequestCD INTEGER NOT NULL PRIMARY KEY,  
AccountCD VARCHAR(6) REFERENCES Abonent(AccountCD)  
ON DELETE CASCADE ON UPDATE CASCADE,  
ExecutorCD INTEGER,  
FailureCD INTEGER,
```



IncomingDate DATE,  
ExecutionDate DATE,  
Executed SMALLINT,  
CONSTRAINT ExecDat CHECK (ExecutionDate  
BETWEEN IncomingDate AND IncomingDate+7));.

Следует обратить внимание, что проверяются значения разных столбцов одной и той же строки – нельзя проверить значения более чем в одной строке. Также нельзя использовать ограничение CHECK, например, для того, чтобы указать зависимость даты выполнения одной ремонтной заявки от даты поступления остальных.

Предложение

[USING [ASC[ENDING] | DESC[ENDING]] INDEX имя\_индекса]

будет рассмотрено позднее при изучении индексов.

#### 4.2.4. Удаление таблицы БД

Для удаления существующей таблицы используется запрос DROP TABLE, который имеет следующий формат:

DROP TABLE базовая\_таблица;.

Следует отметить, что нельзя удалить таблицу в следующих случаях:

- если на ее столбцы ссылаются внешние ключи других таблиц;
- если она используется другими объектами БД (например, представлением);
- если она определена в текущей транзакции, на момент удаления еще не заверченной.

При попытке удалить используемую таблицу выдается сообщение об ошибке с SQLCODE=-607 и сообщением, что удаляемый объект еще находится в использовании.

#### 4.2.5. Изменение определения таблицы

Созданную запросом CREATE TABLE базовую таблицу можно изменить запросом ALTER TABLE. Он поддерживает следующие изменения:

- добавление новых столбцов;
- задание нового ограничения целостности для базовой таблицы;
- определение нового имени для существующего столбца;
- изменение типа данных для существующего столбца;
- изменения порядкового номера столбца в таблице;
- определение для существующего столбца нового значения по умолчанию (замещающего предыдущее значение, если оно было);
- удаление для столбца существующего значения по умолчанию;
- удаление существующего столбца;
- удаление существующего ограничения целостности для базовой таблицы.

Запрос ALTER TABLE имеет следующий формат:

```
ALTER TABLE базовая_таблица <действие1> [, <действие2> ...];,
```

где базовая\_таблица – это имя существующей базовой таблицы БД, определение которой требуется изменить;

<действие> задает действия, которые будут производиться с указанной таблицей, и определяется следующим образом:

```
<действие>::=  
{ADD <определение_столбца>  
 | ADD <тип_ограничения>  
 | ALTER [COLUMN] столбец <изменение>  
 | DROP столбец  
 | DROP CONSTRAINT имя_ограничения},
```

где

```
<изменение>::=  
{ | TO новое_имя_столбца  
 | TYPE {<тип данных> | имя_домена}  
 | POSITION номер_позиции  
 | SET DEFAULT <значение>  
 | DROP DEFAULT}.
```

При использовании конструкции

```
ADD <определение_столбца>
```

в таблицу будет добавлен новый столбец, определенный в соответствии с конструкцией <определение\_столбца>, так же как при создании таблицы.

Использование конструкции

```
ADD <тип_ограничения>
```

приводит к добавлению в таблицу ограничения, определяемого конструкцией <тип\_ограничения>, так же как при создании таблицы.

Например, чтобы добавить столбец с именем memo в таблицу Request, необходимо выполнить следующий запрос ALTER TABLE:

```
ALTER TABLE Request ADD memo VARCHAR(100);.
```

Чтобы для таблицы Request определить ограничения внешних ключей, можно выполнить следующие запросы ALTER TABLE:

```
ALTER TABLE Request ADD FOREIGN KEY (AccountCD)  
REFERENCES Abonent(AccountCD);  
ALTER TABLE Request ADD FOREIGN KEY (ExecutorCD)  
REFERENCES Executor(ExecutorCD);  
ALTER TABLE Request ADD FOREIGN KEY (FailureCD)  
REFERENCES Disrepair(FailureCD);.
```

Однако можно добавить эти внешние ключи для таблицы Request и с помощью одного запроса:

```
ALTER TABLE Request  
ADD FOREIGN KEY (AccountCD) REFERENCES
```

```

Abonent(AccountCD),
ADD FOREIGN KEY (ExecutorCD) REFERENCES
  Executor (ExecutorCD),
ADD FOREIGN KEY (FailureCD) REFERENCES
  Disrepair (FailureCD);

```

Конструкция

```
ALTER столбец TO новое_имя_столбца
```

используется для переименования столбца. Например, можно изменить имя столбца Phone в таблице Abonent следующим образом:

```
ALTER TABLE Abonent ALTER Phone TO HomePhone;
```

Однако попытка переименовать столбец может не удалиться из-за проблемы существования зависимостей (если на столбец существуют ссылки из ограничений или он используется в представлениях, триггерах или хранимых процедурах).

Конструкция

```
ALTER столбец TYPE {<тип данных> | имя_домена}
```

используется для изменения типа данных столбца. Однако существуют некоторые ограничения при изменении типа данных:

- СУБД не позволит произвести такое изменение типа данных, в результате которого могут потеряться данные. Новое определение столбца должно позволять использовать существующие данные. Например, новое количество символов в столбце не может быть меньше наибольшего размера столбца (т.е. количества символов для строковых типов, общего количества разрядов и числа разрядов после запятой для десятичных типов). Например, если попытаться переопределить тип столбца Phone таблицы Abonent с меньшим количеством символов:

```
ALTER TABLE Abonent ALTER Phone Type varchar(10);,
```

то будет выдано сообщение "New size specified for column PHONE must be at least 15 characters" (новый размер, определяемый для столбца PHONE, должен быть не менее 15 символов);

- в СУБД Firebird преобразование числового типа данных в строковый тип требует минимального размера строкового типа, как показано в табл. 4.1;
- преобразование символьных данных в другие типы недопустимо.

**Таблица 4.1.** Минимальное количество символов для числовых преобразований

Тип данных	Минимальная длина символьного типа
BIGINT	19 (или 20 для чисел со знаком)
DECIMAL	20
DOUBLE	22
FLOAT	13
INTEGER	10 (11 для чисел со знаком)
NUMERIC	20 (или 21 для чисел со знаком)
SMALLINT	6

Конструкция

ALTER столбец POSITION номер\_позиции

используется для изменения порядкового номера столбца в таблице.

Конструкция

ALTER столбец SET DEFAULT <значение>

используется для задания значения по умолчанию для столбца или для изменения значения по умолчанию, назначенного для столбца таблицы при ее создании (в том числе для изменения значения по умолчанию, взятого из домена, на котором определен данный столбец). Следует отметить, что изменение значения столбца по умолчанию не оказывает влияния на состояние существующих строк таблицы (даже если в некоторых из них хранится предыдущее значение столбца по умолчанию).

Конструкция

ALTER столбец DROP DEFAULT

используется для удаления значения по умолчанию, определенного для столбца таблицы. Если такое значение не было определено, то выдается соответствующее сообщение. Если столбец определен на домене, у которого существует значение по умолчанию, то после удаления значения по умолчанию, определенного для столбца таблицы, начинает действовать значение по умолчанию домена.

Рассмотрим на примере изменение и удаление значений по умолчанию для столбцов. Допустим, что в созданную ранее (в пункте 4.2.1) таблицу Days добавлена строка, где number = 1, а остальные значения берутся по умолчанию. В результате таблица будет иметь вид, представленный на рис. 4.1.

NUMBER	DAT	EVENT	USR	TEL
1	01.09.1996	<null>	SYSDBA	111111

**Рис. 4.1.** Таблица Days до изменения значений по умолчанию

Например, с помощью следующего запроса удалим в таблице Days значение по умолчанию столбца dat, изменим значение по умолчанию столбца usr на Petrov, а значение по умолчанию столбца tel, взятое из домена Telephone, с '111111' на '999999':

```
ALTER TABLE Days ALTER dat DROP DEFAULT,  
ALTER usr SET DEFAULT 'Petrov',  
ALTER tel SET DEFAULT '999999';
```

После добавления второй строки со значениями по умолчанию таблица Days примет вид, представленный на рис. 4.2.

NUMBER	DAT	EVENT	USR	TEL
1	01.09.1996	<null>	SYSDBA	111111
2	<null>	<null>	Petrov	999999

**Рис. 4.2.** Таблица Days после изменения и вставки

## Конструкция DROP столбец

используется для удаления существующего столбца таблицы. Удаление столбца запросом ALTER TABLE может завершиться неудачей в следующих случаях:

- удаляемый столбец является частью ограничений UNIQUE, PRIMARY KEY или FOREIGN KEY;
- удаляемый столбец используется в предложении CHECK;
- удаляемый столбец является частью выражения, например, в конструкции COMPUTED [BY] (<выражение>) (при создании таблицы);
- на удаляемый столбец ссылается другой объект БД, например представление.

Удаление ограничения таблицы производится при использовании предложения DROP CONSTRAINT запроса ALTER TABLE с указанием имени ограничения. Если при создании таблицы (CREATE TABLE) или изменении ее определения (ALTER TABLE) не задавалось имя ограничения, то удаление такого ограничения становится невозможным.

Например, чтобы удалить ограничение, наложенное на таблицу и имеющее имя Abonent\_not\_as\_Executor, необходимо воспользоваться следующим запросом:

```
ALTER TABLE Abonent DROP CONSTRAINT  
Abonent_not_as_Executor;
```

### 4.3. Индексы

Одним из структурных элементов физической памяти, присутствующим в большинстве современных реляционных СУБД, является индекс. Индекс – это средство, обеспечивающее быстрый доступ к строкам таблицы на основе значений одного или нескольких ее столбцов [19, 23].

В индексе хранятся значения данных и указатели на строки, где эти данные встречаются. При выполнении запроса СУБД сначала определяет список индексов, связанных с данной таблицей. Затем устанавливает, что является более эффективным, просмотреть всю таблицу или для обработки запроса использовать существующий индекс. Если СУБД решает использовать индекс, то поиск ведется сначала по ключевым значениям в индексе, а затем, используя указатели, осуществляется просмотр самих таблиц для дополнительной фильтрации и окончательной выборки требуемых данных.

Поиск осуществляется достаточно быстро, поскольку значения в индексе упорядочены (в убывающем или возрастающем порядке), а сам индекс относительно невелик. Это позволяет найти ключевое значение. Как только ключевое значение найдено, по указателю определяется физическое местоположение связанных с ним данных.

Использование индекса обычно требует меньшего количества обращений к диску, чем последовательное чтение строк в таблице.

Тем не менее, индексирование оправданно далеко не всегда. Следует помнить, что при всяком обновлении данных должны обновляться и индексы.

Таким образом, платой за быстрый поиск является увеличение затрат времени на обновление данных. Кроме того, сами индексы после большого числа обновлений становятся несбалансированными, вследствие чего время поиска по ним возрастает.

В этих условиях при проектировании БД необходимо находить компромисс между требованиями по ускорению поиска данных и по скорости их обновления. Использование индексов, например, для небольших по объему таблиц вообще не оправданно. Если имеется индекс по группе полей, то поиск по первому из полей группы может прямо использовать этот индекс, следовательно, нет смысла делать по нему отдельный индекс. Если поиск по каким-либо полям редок, то построение по ним индекса неэффективно.

В то же время индексирование может дать большой эффект при работе с данными, которые часто используются, но редко меняются, например в таблицах-справочниках. Если часто используются запросы, требующие соединения таблиц по какому-либо полю или группе полей, то от индексирования таблиц по этим полям может быть получен значительный эффект. Кроме того, индекс может быть полезен, если часто выполняется сортировка данных по столбцу или группе столбцов.

Можно сказать, что оптимальный выбор состава и количества индексов зависит и от структуры БД, и от характера ее использования.

В СУБД Firebird автоматически создаются индексы по первичным, внешним ключам таблиц и UNIQUE-ограничениям и их дополнительно создавать не нужно. Такие индексы для ограничений без имени по умолчанию имеют название наподобие RDB\$PRIMARY8, RDB\$FOREIGN13, RDB\$10 и т.д., а для ограничений с именем название индекса совпадает с названием ограничения. Данные в таких индексах по умолчанию располагаются в возрастающем порядке.

Рекомендуется создавать индекс для столбцов, которые часто используются в условиях поиска. В SQL индекс для таких столбцов создается запросом CREATE INDEX, который имеет следующий формат:

```
CREATE [UNIQUE] [ASC | DESC] INDEX имя_индекса
ON базовая_таблица {( <список_столбцов > )
                    | COMPUTED BY ( <выражение > )};,
```

где

- имя\_индекса, задает имя, под которым создаваемый индекс будет определен в БД;
- базовая\_таблица и список\_столбцов, определяют соответственно базовую таблицу и индексируемые столбцы (столбец).

Необязательные параметры запроса CREATE INDEX:

- UNIQUE, предотвращает вставку или обновление повторяющихся значений в индексируемые столбцы;
- ASC, сортирует столбцы в возрастающем порядке (используется по умолчанию);
- DESC, сортирует столбцы в убывающем порядке.

Например, для создания отсортированного по убыванию индекса с именем Month\_Index по полю NachisMonth таблицы NachisSumma необходимо применить следующий запрос:

```
CREATE DESC INDEX Month_Index  
ON NachisSumma (NachisMonth);
```

В некоторых случаях удобно создать составной индекс, т.е. индекс для нескольких столбцов в таблице. Например, если требуется часто осуществлять поиск и сортировку по адресам абонентов, то можно создать составной индекс для полей StreetCD и HouseNO таблицы Abonent. Запрос на создание такого индекса будет выглядеть следующим образом:

```
CREATE INDEX Address_Index  
ON Abonent (StreetCD, HouseNO);
```

Результатом выполнения данного запроса будет создание отсортированного по возрастанию индекса Address\_Index, который обеспечит ускорение поиска и сортировки по адресу абонента.

Существует возможность проиндексировать выражения, часто применяемые в запросах. Индекс для выражения создается с помощью следующей конструкции:

```
COMPUTED BY (<выражение>).
```

Например, можно создать индекс для таблицы Request по выражению, извлекающему значение месяца из столбца IncomingDate, с помощью следующего запроса:

```
CREATE INDEX Ind_1 ON Request  
COMPUTED BY (EXTRACT(MONTH FROM IncomingDate));
```

Данный индекс будет доступен для любого запроса с поиском или сортировкой, если они включают выражение EXTRACT(MONTH FROM IncomingDate).

Индексы для выражений имеют точно такие же характеристики, как и индексы для столбцов, за исключением того, что они не могут быть составными.

Как отмечалось, после описания ограничения таблицы может быть указана следующая конструкция:

```
[USING [ASC[ENDING] | DESC[ENDING]] INDEX имя_индекса].
```

С помощью данной конструкции можно изменить имя индекса, создаваемого по первичному, внешнему или уникальному ключу таблицы, с системного на пользовательское имя и указать нужный порядок расположения данных в индексе. Т.е. можно осуществить переименование автоматически создаваемого индекса и изменение, если требуется, порядка данных в нем.

Например, определим в ранее созданной таблице Days поле number как первичный ключ, задав для связанного с ним индекса имя PK\_NUMBER и порядок данных по убыванию:

```
ALTER TABLE Days ADD PRIMARY KEY (number)  
USING DESC INDEX PK_NUMBER;
```

При изменении порядка сортировки данных в индексах следует помнить, что FOREIGN KEY и соответствующий ему PRIMARY KEY должны использовать одинаковый порядок сортировки в связанных с ними индексах.

Поиск в индексе осуществляется очень быстро, так как индекс отсортирован и его строки очень короткие. К *недостаткам индекса* относится то, что он занимает дополнительное дисковое пространство, и то, что индекс необходимо обновлять каждый раз, когда в таблицу добавляется строка или обновляется проиндексированный столбец таблицы.

Наличие или отсутствие индекса совершенно незаметно для пользователя, обращающегося к таблице. Если для какого-либо столбца создан индекс, то СУБД будет автоматически его использовать.

Чтобы при выполнении запросов деактивизировать или, наоборот, активизировать использование определенного индекса, необходимо воспользоваться запросом ALTER INDEX, который имеет следующий формат:

```
ALTER INDEX имя_индекса {ACTIVE | INACTIVE};
```

При создании любой индекс (как по столбцам ограничений, так и по другим столбцам) автоматически активен. Запрос ALTER INDEX может быть применен для отключения индекса перед добавлением или изменением большого количества строк и устранения при этом дополнительных затрат для поддержки индексов в процессе длительной операции. После этой операции индексирование может быть восстановлено и индексы будут пересозданы.

Удаление индекса с заданным именем производится с помощью запроса DROP INDEX, который имеет следующий формат:

```
DROP INDEX имя_индекса;
```

#### **4.4. Временные таблицы**

Ранее были описаны запросы DDL применительно к постоянным (базовым) таблицам, которые характеризуются тем, что их определение и содержимое существуют в БД до тех пор, пока они не будут удалены явно с помощью соответствующих запросов. Часто в процессе работы возникает необходимость сохранения временных данных (например, промежуточных результатов вычислений в хранимых процедурах). Если хранить такие временные данные в обычных таблицах, то требуются постоянное слежение за содержимым таблиц, а также специфическая организация работы с данными. Более удобно в таком случае использовать временные таблицы.

Рассмотрим, что представляют собой временные таблицы. Сами по себе временные таблицы на самом деле постоянные, то есть при их создании информация сохраняется в системной таблице RDB\$RELATIONS, как и для обычных таблиц. Определение временной таблицы может быть удалено только явно, однако ее содержимое может удаляться или становиться невидимым



(недостижимым) автоматически при достижении определенных условий. Временная таблица определяется следующим образом:

```
CREATE GLOBAL TEMPORARY TABLE имя_временной_таблицы
  (<определение_столбца> [, <определение_столбца>...]
  [, <тип_ограничения>.....])
[ON COMMIT {DELETE | PRESERVE} ROWS];
```

Таким образом, данный синтаксис отличается от синтаксиса создания обычных таблиц фразой GLOBAL TEMPORARY и предложением ON COMMIT.

Глобальные временные таблицы могут быть двух типов: с данными, хранимыми в течение текущего соединения, и с данными, хранимыми только на протяжении выполнения транзакции, использующей временную таблицу. Данные, созданные в разных подключениях (транзакциях), изолированы друг от друга, но метаданные глобальной временной таблицы доступны во всех соединениях и транзакциях.

Тип временной таблицы устанавливается с помощью предложения ON COMMIT.

Если используется ON COMMIT DELETE ROWS, то данные таблицы будут удаляться из БД сразу же после окончания транзакции. Таким образом, таблицы GLOBAL TEMPORARY DELETE хранят записи только до ближайшей команды COMMIT или ROLLBACK, причем не только транзакции, которая их создала, но и любой другой транзакции в этом же подключении. При этом созданные в таблице записи не видны нигде кроме текущей транзакции. После использования как ROLLBACK, так и COMMIT записи во временных таблицах "исчезнут", однако в случае COMMIT все изменения, произведенные над обычными таблицами, будут подтверждены.

Следует отметить, что ON COMMIT DELETE ROWS принимается по умолчанию, если предложение ON COMMIT не задано.

Если создать временную таблицу с помощью следующего запроса:

```
CREATE GLOBAL TEMPORARY TABLE TmpTrans
  (ID INTEGER not null,
  NAME VARCHAR (20),
  CONSTRAINT PK_TmpTrans PRIMARY KEY (ID) )
ON COMMIT DELETE ROWS;;
```

а затем добавить в нее запись

```
INSERT INTO TmpTrans VALUES (1, 'Запись №1');
```

то после вставки не следует подтверждать транзакцию, иначе записи пропадут.

Если теперь выполнить следующий запрос:

```
SELECT * FROM TmpTrans;;
```

то в результате можно получить записи, внесенные во временную таблицу (рис. 4.3).

ID	NAME
1	Запись №1

**Рис. 4.3.** Результат выборки данных из временной таблицы

После выполнения COMMIT повтор последнего запроса вернет в качестве результата пустую таблицу.

Если используется ON COMMIT PRESERVE ROWS, то данные таблицы после окончания транзакции остаются в БД до конца соединения. Т.е. таблицы GLOBAL TEMPORARY PRESERVE хранят записи до отсоединения подключения, в котором они были добавлены, причем их видимость ограничена только этим подключением.

Например, если создать временную таблицу с помощью следующего запроса:

```
CREATE GLOBAL TEMPORARY TABLE TmpConn
  (Id INTEGER NOT NULL,
   Name VARCHAR (20),
   CONSTRAINT PK_TmpConn PRIMARY KEY (ID) )
ON COMMIT PRESERVE ROWS; ,
```

затем добавить в нее запись, например, используя следующий запрос:

```
INSERT INTO TMPCONN
VALUES (1, 'Запись №1 для текущего соединения');
```

и подтвердить транзакцию (COMMIT), то в рамках текущего подключения данная запись будет видна из разных транзакций. Если выполнить еще одно подключение к этой же БД (например, запустив еще один экземпляр IBEExpert) и выполнить тот же самый запрос INSERT, то ошибки "PRIMARY or UNIQUE key constraint" (повтор значения первичного ключа) не возникнет. Как только текущее соединение будет закрыто, вставленные данные пропадут.

Временная таблица, так же как и обычная таблица, может иметь индексы, триггеры, ограничения на уровне столбца и на уровне таблицы. Временные таблицы могут быть связаны между собой отношением родитель-потомок (с помощью задания внешнего ключа).

Однако следует учитывать следующие ограничения:

1) ссылки (ограничения внешнего ключа REFERENCES) между постоянной и временной таблицей запрещены;

2) временная таблица с ON COMMIT PRESERVE ROWS не может иметь ссылку на временную таблицу с ON COMMIT DELETE ROWS.

В заключение можно сказать, что временные таблицы могут быть достаточно полезны для приложений, которые формируют сложные отчеты или производят промежуточные вычисления на сервере. Однако использование временных таблиц может замедлять подключение к БД (если использовались таблицы GLOBAL TEMPORARY DELETE) и отключение от нее (если использовались таблицы GLOBAL TEMPORARY PRESERVE) из-за очистки жесткого диска от версий удаленных записей из таблицы.

## 4.5. Представления

Представление – это виртуальная таблица, созданная на основе запроса из базовой таблицы. Представление, как и реальная (базовая) таблица, содержит строки и столбцы данных, но данные, видимые в представлении, на самом деле являются результатами запроса.

Одной из операций над представлениями является их непосредственное использование с запросами модификации DML: INSERT, UPDATE и DELETE. Если к представлению могут быть успешно применены данные запросы, то оно называется *модифицируемым* (или обновляемым). В противном случае представление является представлением *только для чтения*. Обновление представлений будет подробно рассмотрено позднее после изучения языка манипулирования данными.

В запросах SELECT, INSERT, DELETE и UPDATE на представление можно ссылаться как на обычную таблицу. Это дает возможность определять подмножество данных, необходимых конкретному пользователю (или группе пользователей) в дополнение к ограничению доступа к остальной части данных. Представления используются по следующим причинам:

- они позволяют сделать так, что разные пользователи БД будут видеть ее по-разному;
- с их помощью можно ограничить доступ к данным, разрешая пользователям видеть только некоторые из строк и столбцов таблицы;
- они упрощают доступ к БД, показывая каждому пользователю структуру хранимых данных в наиболее подходящей для него форме.

В SQL представления создаются запросом CREATE VIEW, который имеет следующий формат:

```
CREATE VIEW представление  
[( столбец_представления [, столбец_представления ...]])  
AS <табличный_подзапрос> [WITH CHECK OPTION];
```

Данный запрос создает представление с именем представление.

Параметр WITH CHECK OPTION предотвращает INSERT- или UPDATE-операции над обновляемым представлением, если они нарушают условие отбора строк, определенное в предложении WHERE запроса SELECT (<табличный\_подзапрос>), используемого при определении данного представления.

При необходимости в запросе CREATE VIEW можно задать имя для каждого столбца создаваемого представления, обозначаемое как столбец\_представления. Если указывается список имен столбцов, то он должен содержать столько элементов, сколько столбцов возвращается запросом.

Следует обратить внимание на то, что задаются только имена столбцов; тип данных, длина и другие характеристики берутся из определения столбца в исходной таблице. Если список имен столбцов в запросе CREATE VIEW

отсутствует, то каждый столбец представления получает имя соответствующего столбца запроса. Если в запрос входят вычисляемые столбцы или два столбца с одинаковыми именами, то использование списка имен столбцов является обязательным.

Например, для создания представления Sred\_Summ с полями Month и Summa, показывающего среднее значение начисленных сумм за месяц, следует выполнить следующий запрос:

```
CREATE VIEW Sred_Summ (Mes, Summa)
AS SELECT NachislMonth, AVG (NachislSum)
FROM NachislSumma GROUP BY 1;
```

Результат выполнения запроса

```
SELECT * FROM Sred_Summ;
```

представлен на рис. 4.4.

MES	SUMMA
1	33,58
2	34,58
3	20,61
4	38,38
5	32,09
6	36,98
7	25,26
8	26,45
9	110,24
10	51,36
11	47,92
12	44,23

**Рис. 4.4.** Результат запроса к представлению Sred\_Summ

Ниже приведен пример создания представления с именем Date\_Abonent, которое должно показывать ФИО абонентов и дату подачи ими ремонтных заявок:

```
CREATE VIEW Date_Abonent (Abonent_Name, Data)
AS SELECT Fio, IncomingDate FROM Abonent, Request
WHERE Abonent.AccountCD = Request.AccountCD;
```

В СУБД Firebird 2.1 представления могут использовать все конструкции, допустимые для обычного запроса SELECT. Таким образом, возможно использование конструкций FIRST/SKIP, ROWS, UNION, ORDER BY. Например, для создания представления Max\_Pay с полями AccountCD, Big\_Sum и PayDate, показывающего пять максимальных значений оплаченных сумм, следует выполнить следующий запрос:

```
CREATE VIEW Max_Pay (AccountCD, Big_Sum, PayDate)
AS SELECT FIRST 5 AccountCD, PaySum, PayDate
FROM PaySumma ORDER BY PaySum DESC;
```

Результат выполнения запроса

```
SELECT * FROM Max_Pay;
```

представлен на рис. 4.5.

ACCOUNTCD	BIG_SUM	PAYDATE
115705	250,00	06.10.2000
115705	250,00	03.10.2001
080047	80,00	26.11.1998
443069	80,00	03.10.2001
080047	80,00	21.11.2001

**Рис. 4.5.** Результат запроса к представлению Max\_Pay

Физически представление в БД хранится в виде его определения, т.е. текста того запроса `CREATE VIEW`, который был использован при создании представления. Когда СУБД встречает в SQL-запросе ссылку на представление, она отыскивает его определение, сохраненное в БД. Затем преобразует пользовательский запрос, ссылающийся на представление, в эквивалентный запрос к исходным таблицам представления (заданным в запросе `SELECT` представления) и выполняет этот запрос. Таким образом, СУБД создает иллюзию существования представления в виде отдельной таблицы и в то же время сохраняет целостность исходных данных.

Представление может служить неким «окном» для просмотра данных. Любые изменения в исходных данных (т.е. в данных таблиц, на основе которых создано представление) будут автоматически и мгновенно отображаться в представлении, и наоборот, все изменения, вносимые в данные представления, будут автоматически вноситься в исходные данные и соответственно отображаться в представлении.

По виду запроса, используемого представлением, различают следующие виды представлений:

- горизонтальные;
- вертикальные;
- смешанные;
- сгруппированные;
- соединенные.

*Горизонтальное представление* представляет собой горизонтальное подмножество строк одиночной таблицы и предназначено для ограничения доступа к строкам таблицы. Запрос в таком представлении выбирает все столбцы заданной таблицы, но ограничивает выбор строк указанием условия поиска в предложении `WHERE` запроса `SELECT`. Например, для создания представления, показывающего все ремонтные заявки по неисправности с кодом, равным 1, следует выполнить следующий запрос:

```
CREATE VIEW Failure_Req  
AS SELECT * FROM Request WHERE FailureCD = 1;
```

При выполнении следующего запроса

```
SELECT * FROM Failure_Req;
```

из таблицы Request будут выбраны все ремонтные заявки с кодом неисправности газового оборудования, равным 1. Результат запроса представлен на рис. 4.6.

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
1	005488	1	1	17.12.2001	20.12.2001	1
2	115705	3	1	07.08.2001	12.08.2001	1
5	080270	4	1	31.12.2001	<null>	0
9	136169	2	1	06.11.2001	08.11.2001	1

**Рис. 4.6.** Результат выполнения запроса к представлению Failure\_Req

*Вертикальное представление* представляет собой вертикальное подмножество строк одиночной таблицы и предназначено для ограничения доступа к столбцам таблицы. Запрос в таком представлении выбирает из таблицы требуемые столбцы, а ограничение на выбор строк отсутствует.

Например, если необходимо создать представление Abonent\_Phone, которое должно содержать ФИО абонента и его телефон, следует выполнить следующий запрос:

```
CREATE VIEW Abonent_Phone (Abon_Fio, Abon_Phone)
AS SELECT Fio, Phone FROM Abonent;
```

В этом примере Abon\_Fio и Abon\_Phone - имена столбцов представления. Если не указывать обозначение имен в скобках после имени представления, то столбцы представления получат соответственно имена Fio и Phone.

Следующий запрос:

```
SELECT * FROM Abonent_Phone;
```

даст результат, представленный на рис. 4.7.

ABON_FIO	ABON_PHONE
АКСЕНОВ С.А.	556893
МИЩЕНКО Е.В.	769975
КОНЮХОВ В.С.	761699
ГУЛУПОВА М.И.	214833
СВИРИНА З.А.	350003
СТАРДУБЦЕВ Е.В.	683014
ШМАКОВ С.В.	982222
МАРКОВА В.П.	683301
ДЕНИСОВА Е.К.	680305
ЛУКАШИНА Р.М.	254417
ШУБИНА Т.П.	257842
ТИМОШКИНА Н.Г.	321002

**Рис. 4.7.** Результат выполнения запроса к представлению Abonent\_Phone

*Смешанное представление* представляет собой подмножество строк и столбцов одиночной таблицы. Оно является представлением, разделяющим исходную таблицу как в горизонтальном, так и в вертикальном направлениях.

Пусть необходимо создать представление Nachisl\_Service, отображающее номера лицевого счетов абонентов и начисленные им суммы за услугу газоснабжения с кодом, равным 2. Для решения этой задачи следует выполнить следующий запрос:

```
CREATE VIEW Nachisl_Service  
AS SELECT AccountCD, GazServiceCD, NachislSum  
FROM NachislSumma WHERE GazServiceCD = 2;
```

*Сгруппированное представление* основано на запросе, содержащем предложение GROUP BY и, как следствие, использующем агрегатные функции. Оно выполняет ту же функцию, что и запросы с группировкой определенных столбцов. В них родственные строки данных объединяются в группы, и для каждой группы в таблице результатов запроса создается одна строка, содержащая итоговые данные по этой группе. С помощью сгруппированного представления запрос с группировкой превращается в виртуальную таблицу, к которой в дальнейшем можно обращаться.

В отличие от горизонтальных и вертикальных представлений, каждой строке сгруппированного представления не соответствует какая-то одна строка исходной таблицы. Сгруппированное представление не является просто фильтром исходной таблицы. Оно, в силу использования агрегатных функций, отображает исходную таблицу в виде суммарной информации и требует от СУБД значительного объема вычислений.

Например, если необходимо создать представление Abonent\_All\_Pay с полями AccountCD и All\_Pay, показывающее общую сумму оплат для каждого абонента, то следует выполнить следующий запрос:

```
CREATE VIEW Abonent_All_Pay (AccountCD, All_Pay)  
AS SELECT AccountCD, Sum (PaySum)  
FROM PaySumma GROUP BY AccountCD;.
```

Результат выполнения следующего запроса:

```
SELECT * FROM Abonent_All_Pay;
```

представлен на рис. 4.8.

ACCOUNTCD	ALL_PAY
005488	222,83
015527	84,96
080047	256,88
080270	221,30
080613	114,66
115705	747,95
126112	40,60
136159	16,60
136160	112,60
136169	160,66
443069	195,10
443690	39,47

**Рис. 4.8.** Результат выполнения запроса к представлению Abonent\_All\_Pay

*Соединенное представление* - это подмножество строк и столбцов из нескольких таблиц. Задавая в определении представления, например, двух- или многотабличный запрос, можно создать виртуальную таблицу, данные в которую считываются соответственно из двух или трех различных таблиц. После создания такого представления к нему можно обращаться с помощью однотобличного запроса.

Например, необходимо создать представление с именем `Abonent_Executor`, в котором будут содержаться ФИО абонентов и назначенных по их ремонтным заявкам исполнителей. Для решения данной задачи подойдет следующий запрос:

```
CREATE VIEW Abonent_Executor
  (Abonent_Name, Executor_Name)
AS SELECT A. Fio, E. Fio
  FROM Abonent A, Executor E, Request R
  WHERE R. AccountCD = A.AccountCD AND
        R.ExecutorCD = E. ExecutorCD;
```

Результат выполнения следующего запроса:

```
SELECT * FROM Abonent_Executor;
```

представлен на рис. 4.9.

<b>ABONENT_NAME</b>	<b>EXECUTOR_NAME</b>
АКСЕНОВ С.А.	СТАРОДУБЦЕВ Е.М.
КОНЮХОВ В.С.	СТАРОДУБЦЕВ Е.М.
ЛУКАШИНА Р.М.	СТАРОДУБЦЕВ Е.М.
ШМАКОВ С.В.	СТАРОДУБЦЕВ Е.М.
МИЩЕНКО Е.В.	СТАРОДУБЦЕВ Е.М.
СТАРОДУБЦЕВ Е.В.	СТАРОДУБЦЕВ Е.М.
ШМАКОВ С.В.	СТАРОДУБЦЕВ Е.М.
ДЕНИСОВА Е.К.	БУЛГАКОВ Т.И.
МИЩЕНКО Е.В.	БУЛГАКОВ Т.И.
МИЩЕНКО Е.В.	БУЛГАКОВ Т.И.
МИЩЕНКО Е.В.	ШУБИН В.Г.
ШУБИНА Т.П.	ШУБИН В.Г.
СВИРИНА З.А.	ШУБИН В.Г.
ШУБИНА Т.П.	ШУБИН В.Г.
ТИМОШКИНА Н.Г.	ШЛЮКОВ М.К.
АКСЕНОВ С.А.	ШЛЮКОВ М.К.
МИЩЕНКО Е.В.	ШЛЮКОВ М.К.
ТИМОШКИНА Н.Г.	ШЛЮКОВ М.К.
СТАРОДУБЦЕВ Е.В.	ШКОЛЬНИКОВ С.М.
АКСЕНОВ С.А.	ШКОЛЬНИКОВ С.М.
ДЕНИСОВА Е.К.	ШКОЛЬНИКОВ С.М.

**Рис. 4.9.** Результат выполнения запроса к представлению `Abonent_Executor`



Если представление не используется другими объектами БД, то его можно удалить запросом DROP VIEW, который имеет следующий формат:

DROP VIEW представление;

Если требуется заново создать представление со старым именем, то используется запрос RECREATE VIEW. Синтаксис этого запроса такой же, как и запроса CREATE VIEW. Если представление не существует перед использованием запроса, то его использование эквивалентно использованию CREATE VIEW. Если представление уже существует, то запрос RECREATE VIEW пытается удалить его и создать полностью новый объект (не будет выполнено, если представление используется другим объектом).

Например, создадим заново горизонтальное представление Failure\_Req так, чтобы оно содержало только погашенные заявки по неисправности с кодом, равным 1, с помощью следующего запроса:

```
RECREATE VIEW Failure_Req
AS SELECT * FROM Request
WHERE FailureCD = 1 AND Executed = 1;
```

Результат выполнения следующего запроса:

```
SELECT * FROM Failure_Req;
```

представлен на рис. 4.10.

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
1	005488	1	1	17.12.2001	20.12.2001	1
2	115705	3	1	07.08.2001	12.08.2001	1
9	136169	2	1	06.11.2001	08.11.2001	1

**Рис. 4.10.** Результат выполнения запроса к представлению Failure\_Req

Преимуществами использования представлений являются следующие:

- *обеспечение логической независимости.* Одна из основных задач, которую позволяют решать представления, - обеспечение логической независимости прикладных программ от изменений в структуре базы данных. При изменении структуры изменяются запросы в определениях соответствующих представлений. При этом никаких изменений в программы, работающие с такими представлениями, вносить не нужно;
- *прикладной взгляд на данные.* Представления дают возможность различным пользователям по-разному видеть одни и те же данные. Это особенно ценно при работе различных категорий пользователей с единой интегрированной базой данных. Пользователям предоставляются только интересующие их данные в наиболее удобной для них форме или формате;
- *защита данных.* Представления предоставляют дополнительный уровень защиты данных в таблицах. От определенных пользователей могут быть скрыты некоторые данные, невидимые через предложенное им представление;

- *скрытие сложности данных*. Пользователь работает с представлением как с обычной таблицей, обращаясь к нему по имени, хотя на самом деле представление может представлять собой сложный запрос.

Таким образом, представление предоставляет множество преимуществ, однако имеются и недостатки [25]. Представление – это виртуальная таблица, и, следовательно, при каждом обращении к представлению происходит обработка запроса, затем возврат результата. Выполнение сложных вычислений или наличие множества соединений может приводить к снижению скорости работы.

## 4.6. Комментарии к объектам базы данных

Для сохранения в БД пояснений, относящихся к объектам БД, используются комментарии на объект базы данных. Для создания такого комментария используется запрос COMMENT ON, который имеет следующий формат:

```
COMMENT ON
{DATABASE IS { 'текст' | NULL}
| <базовый_объект> имя_объекта IS { 'текст' | NULL}
| COLUMN {базовая_таблица | представление}. столбец IS { 'текст' | NULL}
| PARAMETER имя_процедуры.имя_параметра IS { 'текст' | NULL} },
```

где

```
<базовый_объект>:: = {DOMAIN | TABLE | VIEW | PROCEDURE
| TRIGGER | EXCEPTION | GENERATOR | SEQUENCE | INDEX | ROLE};
```

'текст' – пояснения к объекту БД. Указание пустой строки " в качестве текста эквивалентно использованию NULL.

В результате действия запроса COMMENT ON в системном каталоге появятся комментарий к описанию объекта БД. Комментарий на саму БД хранится в поле RDB\$DESCRIPTION системной таблицы RDB\$DATABASE. Например, можно добавить комментарий к учебной БД с помощью следующего запроса:

```
COMMENT ON DATABASE IS 'Учебная база данных Абонент';.
```

Следующий запрос добавляет комментарий к домену PKField:

```
COMMENT ON DOMAIN PKField IS 'Домен предназначен для
определения первичных ключей таблиц';.
```

Этот комментарий хранится в поле RDB\$DESCRIPTION системной таблицы RDB\$FIELDS.

Точно так же можно создать комментарии и для других объектов БД и просмотреть их, извлекая нужное поле из системной таблицы с помощью запроса SELECT или открывая необходимую системную таблицу в IBExpert и закладку «Данные» в ней. Так же, открыв эту закладку, можно изменить созданный комментарий непосредственно в RDB\$DESCRIPTION (при щелчке в области данных этого поля) в выпадающем поле для ввода. Для подтверждения изменений следует нажать «ОК» и .

## Контрольные вопросы

1. В каких режимах возможно создание базы данных?
2. Какие SQL-запросы образуют ядро языка определения данных в Firebird?
3. Как выполнить создание таблицы средствами языка SQL?
4. Каким образом может быть задан тип данных столбца при создании таблицы?
5. Какие виды ограничений могут быть заданы при создании таблицы? Чем они отличаются друг от друга?
6. В каких случаях нельзя удалить базовую таблицу из БД?
7. Какие изменения в базовой таблице могут быть сделаны с помощью запроса ALTER TABLE?
8. Что такое индексы? В каких случаях их использование может дать положительный эффект, а в каких индексация не рекомендуется?
9. Как выполнить создание индекса средствами языка SQL?
10. Каким образом создаются временные таблицы в БД? Чем они отличаются от постоянных базовых таблиц?
11. Как создать представление? Какие существуют виды представлений в зависимости от вида используемого запроса? В чем состоят преимущества использования представлений?
12. Как создать комментарий на объект БД?

## 5. Язык манипулирования данными

В предыдущей главе были изучены возможности SQL по определению структур таблиц и представлений БД. В этой главе рассматривается, каким образом осуществляется манипулирование данными в БД.

Изменение данных в БД выполняется с помощью запросов языка DML (Языка Манипулирования Данными). В SQL СУБД Firebird существуют следующие запросы на изменение данных:

- INSERT (вставка данных);
- UPDATE (обновление существующих данных);
- UPDATE OR INSERT (обновление или вставка данных);
- MERGE (обновление или вставка данных таблицы на основе строк, отобранных из другой таблицы);
- DELETE (удаление данных).

При внесении изменений в данные объектов БД следует учитывать правила ссылочной целостности. Эти правила влияют на изменение (или запрещают изменения) первичных и внешних ключей связанных объектов.

Использование вышеуказанных запросов по отношению к базовым таблицам является предметом данной главы. Также в этой главе приводится описание правил манипулирования данными обновляемых представлений.

## 5.1. Добавление новых данных

Добавление данных в определенную таблицу или представление БД осуществляется с помощью запроса INSERT. Существует два вида этого запроса: однострочный и многострочный.

Запрос INSERT можно представить в следующем обобщенном виде:

```
INSERT INTO { базовая_таблица | представление} [(столбец1 [, столбец2] ...)]
{VALUES ({константа1 | переменная1} [,{константа2 | переменная1}]...)}
|<табличный_подзапрос>
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

Предложение INTO определяет таблицу или представление и столбцы, в которые будут вставлены строки. При однократном использовании однострочного запроса INSERT в таблицу или представление можно вставить только одну строку. Если применить запрос INSERT с вложенным запросом SELECT (многострочный запрос INSERT), то с помощью одного запроса можно вставить в таблицу или представление сразу несколько строк из другой таблицы.

Предложение RETURNING позволяет запоминать значения необходимых столбцов в изменяемой строке. Его использование во всех запросах модификации данных (INSERT, UPDATE OR INSERT, UPDATE, DELETE) будет подробнее рассмотрено при изучении процедурного языка.

### 5.1.1. Однострочный запрос INSERT

Формат однострочного запроса INSERT имеет следующий вид:

```
INSERT INTO { базовая_таблица | представление} [(столбец1 [, столбец2] ...)]
VALUES ({константа1 | переменная1} [,{константа2 | переменная2}] ...)}
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

В таблицу или представление вставляется строка со значениями полей, указанными в перечне предложения VALUES (значения), причем *i*-е значение соответствует *i*-му столбцу в списке столбцов. Столбцы, не указанные в списке, заполняются значениями по умолчанию или NULL-значениями, если ограничение, наложенное на столбец, позволяет вставку NULL-значений. Если допустимы NULL-значения и определено значение по умолчанию, вставлено будет значение по умолчанию. Если NULL-значения не допустимы и не определено значение по умолчанию, то запрос на вставку без явного указания значения такого столбца выполнен не будет. Следует также отметить, что если NULL-значения не допустимы, то можно ввести, например, пустую строку для символьного столбца, однако такой подмены делать не рекомендуется.

Рассмотрим вставку строки в таблицу при полном указании списка ее столбцов. Например, для вставки строки в таблицу Abonent можно использовать следующий запрос:

```
INSERT INTO Abonent (AccountCD, StreetCD, HouseNO,  
                    FlatNO, Fio, Phone)  
VALUES ('50000', 8, 1, 1, 'ПЛИТОВ Е.Д.', '556787');
```

Если в списке предложения VALUES указаны значения для всех столбцов модифицируемой таблицы и порядок их перечисления соответствует порядку столбцов в описании таблицы (как в предыдущем примере), то список столбцов в предложении INTO можно опустить. Например, предыдущий оператор вставки строки в таблицу Abonent можно записать следующим образом:

```
INSERT INTO Abonent  
VALUES ('50000', 8, 1, 1, 'ПЛИТОВ Е.Д.', '556787');
```

Если требуется ввести NULL-значение, например, в поле Phone (номер телефона абонента неизвестен), то оно вводится точно так же, как и обычное значение, например:

```
INSERT INTO Abonent  
VALUES ('50000', 8, 1, 1, 'ПЛИТОВ Е.Д.', NULL);
```

В приведенных выше примерах значения, помещаемые в таблицу, располагаются в том порядке, в котором они были созданы запросом CREATE TABLE. Но в некоторых случаях требуется поменять порядок вводимых значений или вводить значения не во все столбцы в таблице. Чтобы удовлетворить этому условию, требуется явно указывать имена и порядок столбцов. Примером этому может служить следующий запрос:

```
INSERT INTO Abonent (AccountCD, StreetCD, Fio)  
VALUES ('50000', 8, 'ПЛИТОВ Е.Д.');
```

### 5.1.2. Многострочный запрос INSERT

Для вставки в одну таблицу или представление нескольких строк из другой таблицы следует использовать так называемый многострочный запрос INSERT, формат которого имеет следующий вид:

```
INSERT INTO { базовая_таблица | представление} [(столбец1 [, столбец2] ...)]  
  <табличный_подзапрос>  
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

В этом формате запроса INSERT сначала выполняется подзапрос SELECT, представляющий собой <табличный\_подзапрос>, с помощью которого в памяти формируется рабочая таблица, а затем строки рабочей таблицы загружаются в модифицируемую таблицу или представление. При этом *i*-й столбец рабочей таблицы соответствует *i*-му столбцу в списке столбцов модифицируемой таблицы или представления. При использовании многострочного запроса INSERT требуется следить за тем, чтобы тип и количество значений,

возвращаемых подзапросом, совпадали с типом и количеством столбцов в модифицируемой таблице.

Пусть имеется пустая таблица Fio со столбцами abonent\_name и executor\_name, определёнными на типе данных VARCHAR(20). В эту таблицу необходимо поместить фамилии абонентов и исполнителей каждой из ремонтных заявок. Для этого можно применить следующий запрос:

```
INSERT INTO Fio (Abonent_name, Executor_name)
SELECT A.Fio, E.Fio
FROM Abonent A, Executor E, Request R
WHERE R.AccountCD = A.AccountCD AND
R.ExecutorCD = E.ExecutorCD;.
```

В предложении FROM подзапроса SELECT может быть указана не базовая таблица, а представление. Таким образом, можно вставлять данные из представления в базовую таблицу. Например, если ранее было создано представление Abonent\_Executor со столбцами abonent\_name и executor\_name, то предыдущий запрос можно переписать следующим образом:

```
INSERT INTO Fio
SELECT Abonent_name, Executor_name
FROM Abonent_Executor;.
```

Содержание таблицы Fio после вставки будет совпадать с данными, приведенными ранее на рис. 4.8, так как фактически в таблицу Fio внесены все данные ранее созданного представления Abonent\_Executor.

В подзапросах, используемых в многострочном запросе INSERT, можно использовать предложение UNION. Пусть имеется таблица Phone со столбцами Abonent\_Fio, Old\_Tel и New\_Tel, определёнными на типе данных VARCHAR(20), VARCHAR(15) и VARCHAR(15) соответственно. Известно, что в номерах телефона, где первые две цифры были 25, эти цифры стали теперь равны 14, а где были 68 – стали равны 57. В таблицу Phone необходимо поместить фамилии абонентов, у которых изменился номер телефона, а также старые и новые номера телефонов. Для этого можно применить следующий запрос:

```
INSERT INTO Phone (Abonent_Fio, Old_Tel, New_Tel)
SELECT Fio, Phone, '14' || TRIM (LEADING '25' FROM Phone)
FROM Abonent
WHERE SUBSTRING(Phone FROM 1 FOR 2)= '25'
UNION
SELECT Fio, Phone, '57' || TRIM (LEADING '68' FROM Phone)
FROM Abonent
WHERE SUBSTRING(Phone FROM 1 FOR 2)= '68';.
```

Содержание таблицы Phone после вставки представлено на рис. 5.1.

ABONENT_FIO	OLD_TEL	NEW_TEL
ДЕНИСОВА Е.К.	680305	570305
ЛУКАШИНА Р.М.	254417	144417
МАРКОВА В.П.	683301	573301
СТАРОДУБЦЕВ Е.В.	683014	573014
ШУБИНА Т.П.	257842	147842

**Рис. 5.1.** Таблица Phone после вставки

Ограничения.

1. В подзапросе нельзя использовать предложение ORDER BY.
2. В качестве значений, возвращаемых подзапросом, можно использовать агрегатные функции, но при этом нужно обеспечить совпадение типа и количества значений, возвращаемых подзапросом.
3. В предложении FROM подзапроса нельзя использовать ту же самую таблицу, в которую запросом INSERT производится вставка строк.

## 5.2. Обновление существующих данных

Обновление значений, как отдельных строк, так и всей таблицы или представления БД, осуществляется запросом UPDATE. Обновление с помощью запроса UPDATE может быть позиционированным (выполняется только над одной строкой) и поисковым (выполняется над нулевым или большим количеством строк). Позиционированное изменение может появиться только в контексте текущей операции с курсором в модуле на PSQL, в то время как поисковое изменение появляется во всех остальных случаях. Следует отметить, что поисковое изменение может эмулировать позиционированное изменение, если в предложении WHERE задано условие, уникально определяющее строку (например, использующее первичный ключ таблицы). Запрос UPDATE может анализировать информацию из других таблиц БД, используя запрос SELECT (вложенный запрос).

### 5.2.1. Простой запрос UPDATE

Простой запрос UPDATE имеет следующий формат:

```
UPDATE { базовая_таблица | представление}
SET столбец1 = {<значение1> | NULL} [, столбец2 = {<значение2> | NULL} ] ...
[WHERE <условие_поиска> | WHERE CURRENT OF имя_курсора]
[RETURNING <список_столбцов> [INTO <список_переменных>]];
где
<значение> ::= { столбец | константа | <выражение> | функция}.
```

Запрос UPDATE содержит предложение SET, в котором указывается на изменение, которое нужно сделать для определенного столбца. Предложение

SET может включать столбцы лишь из обновляемой таблицы, т.е. значение одного или нескольких столбцов модифицируемой таблицы может быть заменено другим значением. В качестве константы в контексте текущего соединения клиента может использоваться переменная USER. Также можно установить для столбца NULL-значение.

При отсутствии предложения WHERE обновляются значения указанных столбцов во всех строках модифицируемой таблицы. Например, чтобы установить для всех абонентов номер телефона 982223, необходимо выполнить следующий запрос:

```
UPDATE Abonent SET Phone = 982223;
```

Чтобы изменить не все строки, а лишь те, которые удовлетворяют определенным условиям, необходимо задать это условие в предложении WHERE запроса UPDATE. Предположим, что абонент Шубина Т.П. (номер лицевого счета равен '080047') вышла замуж и у неё изменилась фамилия на Серова Т.П. Для внесения в таблицу Abonent соответствующих изменений нужно выполнить следующий запрос:

```
UPDATE Abonent SET Fio = 'СЕРОВА Т.П.'
WHERE AccountCD = '080047';.
```

Таблица Abonent после обновления представлена на рис. 5.2.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
005488	3	4	1	АКСЕНОВ С.А.	556893
115705	3	1	82	МИЩЕНКО Е.В.	769975
015527	3	1	65	КОНЮХОВ В.С.	761699
443690	7	5	1	ТУЛУПОВА М.И.	214833
136159	7	39	1	СВИРИНА З.А.	350003
443069	4	51	55	СТАРОДУБЦЕВ Е.В.	683014
136160	4	9	15	ШМАКОВ С.В.	982222
126112	4	7	11	МАРКОВА В.П.	683301
136169	4	7	13	ДЕНИСОВА Е.К.	680305
080613	8	35	11	ЛУКАШИНА Р.М.	254417
080047	8	39	36	СЕРОВА Т.П.	257842
080270	6	35	6	ТИМОШКИНА Н.Г.	321002

**Рис. 5.2.** Таблица Abonent после обновления

Позиционированные обновления, использующие предложение WHERE CURRENT OF имя\_курсора, будут подробно рассмотрены при изучении процедурного SQL.

### 5.2.2. Запрос UPDATE с подзапросом

В UPDATE можно в предложении WHERE использовать вложенные запросы. Также допускается использование вложенного запроса в предложении SET. Запрос UPDATE с подзапросом имеет следующий формат:



```
UPDATE { базовая_таблица | представление}
SET столбец1 [, столбец2] ... = (<подзапрос>)
[WHERE <условие_поиска_с_подзапросом> | WHERE CURRENT OF имя_курсора]
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

В предложении SET <подзапрос> представляет собой <скалярный\_подзапрос>, если указан один столбец для обновления, и <подзапрос\_столбца>, если указано несколько столбцов. В предложении WHERE <подзапрос> может представлять собой <скалярный\_подзапрос>, <подзапрос\_столбца> или <табличный\_подзапрос> в зависимости от вида условия поиска (синтаксис условия поиска совпадает с описанным ранее при изучении вложенных запросов).

Рассмотрим использование подзапроса в предложении WHERE запроса UPDATE, которое дает возможность отбирать строки для обновления на основе информации из других таблиц. Например, для уменьшения на 10 руб. сумм, начисленных за услуги газоснабжения абонентам, имеющим не более одной заявки на ремонт газового оборудования, нужно выполнить следующий запрос:

```
UPDATE NachislSumma N
SET NachislSum = NachislSum - 10
WHERE 2 > (SELECT COUNT (*) FROM Request R
           WHERE R.AccountCD = N.AccountCD);
```

Фрагмент таблицы NachislSumma после обновления представлен на рис. 5.3.

NACHISLFACTCD	ACCOUNTCD	GAZSERVICECD	NACHISLSUM	NACHISLMONTH	NACHISLYEAR
...	.....	.....	.....	...	...
10	080613	2	46,00	6	2001
...	.....	.....	.....	...	...
20	015527	1	18,32	7	1998
.....	.....	.....	.....	.....	.....
22	080613	1	0,60	9	1998
.....	.....	.....	.....	.....	.....
24	015527	1	28,32	4	1999
.....	.....	.....	.....	.....	.....
26	080613	1	2,60	8	2000
.....	.....	.....	.....	.....	.....
29	136159	1	-1,70	8	1999
.....	.....	.....	.....	.....	.....
32	443690	1	7,80	6	1998
.....	.....	.....	.....	.....	.....
34	126112	1	5,30	8	2000
.....	.....	.....	.....	.....	.....
36	080613	1	2,60	4	1998
.....	.....	.....	.....	.....	.....
40	015527	1	8,32	2	1998
41	443690	1	11,67	3	1999
42	080613	1	12,86	4	2000
.....	.....	.....	.....	.....	.....
46	126112	1	15,30	8	2001
.....	.....	.....	.....	.....	.....
48	136159	1	-1,70	10	1998

**Рис. 5.3.** Таблица NachislSumma после уменьшения начисленных сумм

Подзапрос, использованный в данном запросе UPDATE, является связанным, так как в нем используется поле AccountCD внешней обновляемой таблицы (условие R.AccountCD = N.AccountCD). При выполнении обновления сначала выбирается строка из таблицы NachislSumma, затем выполняется вложенный запрос, возвращая количество ремонтных заявок для выбранного абонента, а уже затем, в случае удовлетворения условию в предложении WHERE запроса UPDATE, происходит обновление текущей строки таблицы NachislSumma. Обновление происходит последовательно для каждой строки таблицы NachislSumma без возврата к извлечению уже просмотренных строк.

Приведем еще один пример с использованием соотнесенного вложенного запроса. Пусть требуется уменьшить на 10 руб. суммы, начисленные за сентябрь 2001, абонентам, заплатившим до 15 сентября 2001 года за август 2001 (произвели оплату за август 2001 до 15.09.2001) сумму, меньше средней суммы всех своих платежей. Для решения этой задачи можно использовать следующий запрос:

```
UPDATE NachislSumma SET NachislSum = NachislSum - 10
WHERE NachislYear = 2001 AND NachislMonth = 9
      AND AccountCD IN
      (SELECT AccountCD
       FROM PaySumma A
       WHERE PayYear = 2001 AND PayMonth = 8 AND
            PayDate < '15.09.2001' AND PaySum <
            (SELECT AVG(PaySum) FROM PaySumma B
             GROUP BY AccountCD
             HAVING A.AccountCD = B.AccountCD));
```

Фрагмент таблицы NachislSumma после обновления представлен на рис. 5.4.

NACHISLFACTCD	ACCOUNTCD	GAZSERVICECD	NACHISLSUM	NACHISLMONTH	NACHISLYEAR
.....	.....	....	.....	....	.....
5	115705	2	240,00	9	2001
.....	.....	....	.....	....	.....
17	443069	2	70,00	9	2001
.....	.....	....	.....	....	.....
47	443069	1	28,32	9	2001
.....	.....	....	.....	....	.....

**Рис. 5.4.** Содержание таблицы NachislSumma после обновления

В предложении FROM подзапроса в запросе UPDATE может присутствовать имя обновляемой таблицы. В этом случае предполагается, что используется то состояние обновляемой таблицы, которое она имела до обновления.

Например, требуется уменьшить в 2 раза суммы начислений за услугу газоснабжения с кодом 2, которые превышают среднюю сумму начислений по всем абонентам за эту же услугу за декабрь 2001 года. Для этого необходимо применить следующий запрос UPDATE с подзапросом:

```

UPDATE NachisSumma SET NachisSum = NachisSum / 2
WHERE NachisSum > (SELECT AVG (NachisSum)
FROM NachisSumma
WHERE NachisMonth = 12 AND
NachisYear = 2001 AND
GazServiceCD=2)
AND GazServiceCD=2;

```

Фрагмент таблицы NachisSumma после обновления представлен на рис. 5.5.

NACHISFACTCD	ACCOUNTCD	GAZSERVICECD	NACHISLSUM	NACHISLMONTH	NACHISLYEAR
19	005488	2	29,35	12	2001
.....	.....	.....	.....	.....	.....
3	005488	2	28,00	4	1999
.....	.....	.....	.....	.....	.....
5	115705	2	125,00	9	2001
.....	.....	.....	.....	.....	.....
1	136160	2	28,00	1	1999
.....	.....	.....	.....	.....	.....
7	080047	2	40,00	10	1998
8	080047	2	40,00	10	2001
.....	.....	.....	.....	.....	.....
10	080613	2	28,00	6	2001
11	115705	2	125,00	9	2000
12	115705	2	29,35	8	2001
16	136169	2	29,35	11	2001
17	443069	2	40,00	9	2001
.....	.....	.....	.....	.....	.....

**Рис. 5.5.** Таблица NachisSumma после обновления

В предыдущем примере использован вложенный запрос, который был успешно выполнен. Это произошло потому, что сначала один раз выполнялся вложенный запрос, возвратив среднее значение сумм начислений за декабрь 2001 года, а затем произошло обновление таблицы NachisSumma. В результате выполнения запроса в таблице NachisSumma 11 записей были обновлены.

Аналогично возможно использование соотнесенного вложенного запроса, относящегося к той же самой таблице, которая указана в качестве обновляемой. Рассмотрим пример такого соотнесенного подзапроса в предложении SET. Например, для замены дат выполнения ремонтных заявок, относящихся к каждому абоненту, на наиболее позднюю дату их регистрации можно использовать следующий запрос:

```

UPDATE Request
SET ExecutionDate =
(SELECT MAX (R.IncomingDate) FROM Request R
WHERE R.AccountCD = Request.AccountCD);

```

Таблица Request после обновления представлена на рис. 5.6.

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
1	005488	1	1	17.12.2001	17.12.2001	1
2	115705	3	1	07.08.2001	28.12.2001	1
3	015527	1	12	28.02.1998	28.02.1998	0
5	080270	4	1	31.12.2001	31.12.2001	0
6	080613	1	6	16.06.2001	16.06.2001	1
7	080047	3	2	20.10.1998	11.10.2001	1
9	136169	2	1	06.11.2001	06.11.2001	1
10	136159	3	12	01.04.2001	01.04.2001	0
11	136160	1	6	12.01.1999	18.05.2001	1
12	443069	5	4	08.08.2001	13.09.2001	1
13	005488	5	8	04.09.2000	17.12.2001	1
14	005488	4	6	04.04.1999	17.12.2001	1
15	115705	4	5	20.09.2000	28.12.2001	1
16	115705	2	3	28.12.2001	28.12.2001	0
17	115705	1	5	15.08.2001	28.12.2001	1
18	115705	2	3	28.12.1999	28.12.2001	1
19	080270	4	8	17.12.2001	31.12.2001	1
20	080047	3	2	11.10.2001	11.10.2001	1
21	443069	1	2	13.09.2001	13.09.2001	1
22	136160	1	7	18.05.2001	18.05.2001	1
23	136169	5	7	07.05.2001	06.11.2001	1

**Рис. 5.6.** Таблица Request после обновления

После выполнения обновления у каждого абонента стала одинаковой дата выполнения всех ремонтных заявок (она стала совпадать с датой поступления последней заявки, поданной этим абонентом). Например, от абонента с номером лицевого счета '005488' поступило 3 ремонтных заявки: первая – 04.04.1999, вторая – 04.09.2000 и последняя – 17.12.2001. Дата выполнения всех трех заявок после выполнения запроса стала равной 17.12.2001.

### 5.3. Обобщенное обновление и добавление данных

Помимо запросов UPDATE и INSERT существует обобщенный запрос UPDATE OR INSERT, который предоставляет возможность изменять или вставлять запись в зависимости от того, существует она в целевой таблице или нет. Такой запрос имеет следующий синтаксис:

```
UPDATE OR INSERT INTO { базовая_таблица | представление} [(<список_столбцов>)]
VALUES ({константа1 | переменная1} [, {константа2 | переменная2}] ...)
[MATCHING <список_столбцов>]
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

Например, требуется добавить исполнителя ПЕТРОВ А.С. с кодом 1, причем если исполнитель с таким кодом уже существует, то следует изменить его ФИО на ПЕТРОВ А.С. Запрос будет выглядеть следующим образом:

```
UPDATE OR INSERT INTO Executor VALUES (1,'ПЕТРОВ А.С.);
```

Результат выполнения запроса представлен на рис. 5.7.

EXECUTORCD	FIO
1	ПЕТРОВ А.С.
2	БУЛГАКОВ Т.И.
3	ШУБИН В.Г.
4	ШЛЮКОВ М.К.
5	ШКОЛЬНИКОВ С.М.

**Рис. 5.7.** Результат работы запроса UPDATE OR INSERT

В случае если бы исполнителя с кодом 1 не существовало в таблице Executor, в нее была бы вставлена новая строка.

Предложение MATCHING используется для указания столбцов в таблице, значения в которых следует сопоставить с соответствующими значениями в списке VALUES. По умолчанию сопоставляются значения в столбцах первичных ключей.

**Примечание.** Если предложение MATCHING не используется, то в целевой таблице должен обязательно существовать первичный ключ.

Например, предыдущий запрос с предложением MATCHING может быть записан как

```
UPDATE OR INSERT INTO Executor
VALUES (1,'ПЕТРОВ А.С.') MATCHING (ExecutorCD);,
```

что подразумевается по умолчанию.

Рассмотрим пример, когда может быть полезным использование предложения MATCHING. Например, требуется внести информацию о ремонтной заявке с кодом 25, поданной 17 декабря 2001 года абонентом с номером лицевого счета '005488' и выполненной 22 декабря 2001 года. Запрос может выглядеть следующим образом:

```
UPDATE OR INSERT INTO Request
(RequestCD, AccountCD, IncomingDate, ExecutionDate)
VALUES (25,'005488','17.12.2001','22.12.2001');
```

В процессе выполнения запроса происходит сравнение значения 25 со значениями столбца RequestCD (первичный ключ) в таблице Request. Совпадения не обнаруживается, поэтому происходит вставка новой строки в таблицу Request. Фрагмент таблицы Request после выполнения запроса представлен на рис. 5.8.

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
...	...	...	...	...	...	...
25	005488	<null>	<null>	17.12.2001	22.12.2001	<null>

**Рис. 5.8.** Последняя запись в таблице Request после добавления

Предположим теперь, что вставляются те же данные, но надо учесть, что если информация о заявке абонента с лицевым счетом '005488' от 17 декабря 2001 года уже зарегистрирована, то следует обновить соответствующую строку. Тогда следует использовать следующий запрос:

```

UPDATE OR INSERT INTO Request
  (RequestCD, AccountCD, IncomingDate, ExecutionDate)
VALUES (25,'005488','17.12.2001','22.12.2001')
MATCHING (AccountCD, IncomingDate);

```

Фрагмент таблицы Request после выполнения запроса представлен на рис. 5.9.

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
25	005488	1	1	17.12.2001	22.12.2001	1
...	...	...	...	...	...	...

**Рис. 5.9.** Обновленная первая строка в таблице Request

В процессе выполнения данного запроса значения первичных ключей не сравниваются, так как в предложении MATCHING явно указано, что надо сравнить значения в столбцах AccountCD и IncomingDate с соответствующими значениями из списка VALUES. Обнаруживается совпадение в первой строке таблицы Request и происходит ее обновление.

В списке значений предложения VALUES могут быть указаны не константы, а переменные, если запрос используется в модуле процедурного SQL .

Следует также отметить, что для выполнения данного запроса у пользователя должны быть права как на вставку, так и на изменение записей в таблице или представлении (управление доступом к ресурсам будет подробно описано далее).

## 5.4. Слияние данных

Для выбора строк из одной таблицы с целью обновления или вставки строк в другую таблицу используется запрос MERGE. Выбор действия – обновить строку или добавить ее – определяется в зависимости от условия. Запрос MERGE фактически представляет собой объединение запросов INSERT и UPDATE и позволяет избежать их многократного использования.

Запрос имеет следующий синтаксис:

```

MERGE
INTO <целевая_таблица> [ [AS] псевдоним ]
USING <исходная_таблица> [ [AS] псевдоним ] ON <условие_соединения>
[ WHEN MATCHED THEN
  UPDATE SET столбец1 = <выражение1> [, столбец2 = <выражение2>] ... ]
[ WHEN NOT MATCHED THEN
  INSERT [ ( <список_столбцов> ) ]
  VALUES ( {константа1 | переменная1} [, {константа2 | переменная2}] ... ) ];,

```

где

<целевая\_таблица>:: = { базовая\_таблица | представление},

<исходная\_таблица>:: = { базовая\_таблица | представление | <производная\_таблица> }.

Запрос MERGE работает следующим образом. Выбирается первая строка из исходной таблицы и проверяется, существует ли такая строка целевой таблицы, что на ней становится истинным <условие\_соединения>. Если да – то производится обновление этой строки целевой таблицы согласно запросу UPDATE, указанному в предложении WHEN MATCHED. Если таких строк целевой таблицы несколько, все они обновляются. Если для текущей строки исходной таблицы <условие\_соединения> возвращает FALSE для всех строк целевой таблицы, то в целевую таблицу вставляется строка согласно запросу INSERT, указанному в предложении WHEN NOT MATCHED. В одном запросе могут быть определены предложения WHEN MATCHED и WHEN NOT MATCHED одновременно или одно из них (обязательно).

Описанная выше процедура повторяется для всех строк исходной таблицы.

Допустим, требуется выбрать абонентов, проживающих на улице с кодом 7, и у всех поданных ими ремонтных заявок изменить код исполнителя на 5. Если у выбранного абонента нет зарегистрированных заявок, следует внести информацию в таблицу ремонтных заявок. Запрос на слияние данных будет выглядеть следующим образом:

```

MERGE
INTO Request R
USING (SELECT * FROM Abonent WHERE StreetCD = 7) Ab
ON (R.AccountCD = Ab.AccountCD)
WHEN MATCHED THEN
    UPDATE SET ExecutorCD = 5
WHEN NOT MATCHED THEN
    INSERT (RequestCD, AccountCD, ExecutorCD)
    VALUES (25, Ab.AccountCD, 5);

```

Фрагмент таблицы Request после выполнения запроса представлен на рис. 5.10.

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
...	...	...	...	...	...	...
10	136159	5	12	01.04.2001	03.04.2001	0
...	...	...	...	...	...	...
25	443690	5	<null>	<null>	<null>	<null>

**Рис. 5.10.** Обновленные и добавленные строки в таблице Request

Производной таблицей, определенной в предложении USING, возвращаются 2 строки, содержащие информацию об абонентах с номерами лицевых счетов '136159' и '443690', проживающих на улице с кодом 7. Так как у абонента со счетом '136159' зарегистрирована ремонтная заявка с кодом 10, то значение столбца ExecutorCD в данной строке изменяется на 5. У абонента со счетом

'443690' нет ни одной зарегистрированной заявки, поэтому строка, соответствующая данному абоненту, добавляется в таблицу Request.

Следует отметить, что если на улице с кодом 7 проживало бы несколько абонентов, не подававших ремонтные заявки, то запрос не выполнялся бы из-за повторяющихся вставляемых значений в столбец первичного ключа (RequestCD=25). Для предотвращения такой ситуации следует использовать генераторы последовательностей для получения очередного значения первичного ключа (генераторы будут рассмотрены при изучении процедурного языка).

## 5.5. Удаление существующих данных

Удаление всех данных или отдельных строк из таблицы или представления осуществляется запросом DELETE. Существует два вида этого запроса: простой запрос DELETE и запрос DELETE с подзапросом. Удаление данных с помощью запроса DELETE может быть позиционированным и поисковым (аналогично описанному ранее обновлению данных с помощью запроса UPDATE).

### 5.5.1. Простой запрос DELETE

Простой запрос DELETE позволяет удалить содержимое всех строк указанной таблицы или тех ее строк, которые выделяются условием поиска предложения WHERE, и имеет следующий формат:

```
DELETE FROM {базовая_таблица | представление}
[WHERE <условие_поиска> | WHERE CURRENT OF имя_курсора]
[RETURNING <список_столбцов> [INTO <список_переменных>]];
```

С помощью DELETE можно удалять только отдельные строки, а не индивидуальные значения полей, поэтому параметр поля является недоступным. Например, для удаления всего содержимого таблицы Abonent можно выполнить следующий запрос:

```
DELETE FROM Abonent;
```

Обычно требуется удалить только определенные строки из таблицы или представления. Чтобы определить, какие строки будут удалены, необходимо использовать условие поиска, как это делается для запроса SELECT. Например, чтобы удалить из таблицы Abonent абонента с номером лицевого счета, равным '005488', требуется выполнить следующий запрос:

```
DELETE FROM Abonent WHERE AccountCD = '005488';
```

Однако соответствующая запись из таблицы Abonent удалена не будет, так как имеются записи для данного абонента в дочерних таблицах (NachislSumma, PaySumma, Request).



Можно использовать DELETE с условием поиска, которое затрагивает несколько строк, например следующим образом:

```
DELETE FROM NachisSumma WHERE NachisYear = 1998;
```

Позиционированные удаления данных, использующие предложение WHERE CURRENT OF имя\_курсора, будут подробно рассмотрены при изучении процедурного языка SQL.

### 5.5.2. Запрос DELETE с подзапросом

В предложении WHERE запроса DELETE можно использовать вложенные запросы, которые могут быть как простыми, так и коррелированными. Использование подзапросов в DELETE аналогично использованию таковых в SELECT.

Рассмотрим два примера использования простого подзапроса в предложении WHERE запроса DELETE.

Для удаления ремонтных заявок, исполнителями которых не являются исполнители ремонтных заявок, зарегистрированных 17 декабря 2001 года, необходимо выполнить следующий запрос:

```
DELETE FROM Request WHERE ExecutorCD NOT IN
(SELECT ExecutorCD FROM Request A
WHERE IncomingDate = '17.12.2001');
```

Таблица Request после удаления представлена на рис. 5.11.

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
1	005488	1	1	17.12.2001	20.12.2001	1
3	015527	1	12	28.02.1998	08.03.1998	0
5	080270	4	1	31.12.2001	<null>	0
6	080613	1	6	16.06.2001	24.06.2001	1
11	136160	1	6	12.01.1999	12.01.1999	1
14	005488	4	6	04.04.1999	13.04.1999	1
15	115705	4	5	20.09.2000	23.09.2000	1
17	115705	1	5	15.08.2001	06.09.2001	1
19	080270	4	8	17.12.2001	27.12.2001	1
21	443069	1	2	13.09.2001	14.09.2001	1
22	136160	1	7	18.05.2001	25.05.2001	1

**Рис. 5.11.** Таблица Request после удаления

Запрос DELETE удаляет все строки со значением поля ExecutorCD, входящим во множество значений, формируемых вложенным запросом. В итоге из таблицы Request будут удалены все строки, в которых значения поля ExecutorCD не равны 1 и 4 (исполнителями ремонтных заявок, зарегистрированных 17 декабря 2001 года, являются исполнители с кодами, равными 1 и 4).

Пусть необходимо удалить сведения о ремонтных заявках абонента Мищенко Е.В. Запрос на удаление соответствующих заявок можно записать в следующем виде:

```
DELETE FROM Request WHERE AccountCD IN
(SELECT AccountCD FROM Abonent
WHERE Fio = 'МИЩЕНКО Е.В.);
```

Таблица Request после удаления представлена на рис. 5.12

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
1	005488	1	1	17.12.2001	20.12.2001	1
3	015527	1	12	28.02.1998	08.03.1998	0
5	080270	4	1	31.12.2001	<null>	0
6	080613	1	6	16.06.2001	24.06.2001	1
7	080047	3	2	20.10.1998	24.10.1998	1
9	136169	2	1	06.11.2001	08.11.2001	1
10	136159	3	12	01.04.2001	03.04.2001	0
11	136160	1	6	12.01.1999	12.01.1999	1
12	443069	5	4	08.08.2001	10.08.2001	1
13	005488	5	8	04.09.2000	05.12.2000	1
14	005488	4	6	04.04.1999	13.04.1999	1
19	080270	4	8	17.12.2001	27.12.2001	1
20	080047	3	2	11.10.2001	11.10.2001	1
21	443069	1	2	13.09.2001	14.09.2001	1
22	136160	1	7	18.05.2001	25.05.2001	1
23	136169	5	7	07.05.2001	08.05.2001	1

**Рис. 5.12.** Таблица Request после удаления ремонтных заявок абонента Мищенко Е.В.

Вложенные запросы в предложении WHERE могут иметь *несколько уровней вложенности*. Они могут также содержать внешние ссылки на целевую таблицу запроса DELETE. При этом единственное ограничение на применение подзапросов заключается в том, что *целевую таблицу нельзя указывать в предложении FROM вложенного запроса* независимо от уровня вложенности. Это предотвращает ссылки из вложенных запросов на целевую таблицу (часть строк которой может быть удалена), за исключением внешних ссылок на строку, проверяемую в данный момент на соответствие условию поиска запроса DELETE.

Рассмотрим использование коррелированного подзапроса в предложении WHERE запроса DELETE, т.е. подзапроса, который содержит внешнюю ссылку на текущую строку таблицы, из которой удаляются данные. Например, для удаления всех сведений об оплате услуг газоснабжения абонентами, проживающими на улице Татарской, можно использовать следующий запрос:

```
DELETE FROM PaySumma P WHERE EXISTS
(SELECT * FROM Abonent A, Street S
WHERE S.StreetNM = 'ТАТАРСКАЯ УЛИЦА' AND
A.StreetCD = S. StreetCD AND
P.AccountCD = A. AccountCD);
```

Следует обратить внимание на то, что в этом примере при проверке условия P.AccountCD = A.AccountCD внутренний запрос ссылается к строке таблицы PaySumma, проверяемой в данный момент. Это означает, что подзапрос будет выполняться отдельно для каждой строки таблицы PaySumma. В результате выполнения запроса из таблицы PaySumma будет удалено 15 записей об оплатах абонентов с номерами лицевого счетов '126112', '136160', '136169' и '443069' (именно они проживают на улице Татарской).

Следует отметить, что для этого примера имеется другой способ выполнить те же самые действия в следующем виде:

```
DELETE FROM PaySumma P
WHERE 'ТАТАРСКАЯ УЛИЦА' IN
(SELECT StreetNM FROM Street S, Abonent A
WHERE A.StreetCD = S. StreetCD AND
P.AccountCD = A. AccountCD);
```

В качестве подзапроса в предложении WHERE запроса DELETE можно использовать запрос SELECT, который внутри себя содержит *соотнесенный подзапрос*.

Например, требуется удалить ремонтные заявки тех исполнителей, которые выполнили менее 4 ремонтных заявок. Запрос будет выглядеть следующим образом:

```
DELETE FROM Request
WHERE ExecutorCD IN
(SELECT ExecutorCD FROM Request A
WHERE 4 > (SELECT COUNT(ExecutionDate) FROM Request B
WHERE A.ExecutorCD=B.ExecutorCD));
```

Таблица Request после удаления представлена на рис. 5.13.

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
1	005488	1	1	17.12.2001	20.12.2001	1
2	115705	3	1	07.08.2001	12.08.2001	1
3	015527	1	12	28.02.1998	08.03.1998	0
6	080613	1	6	16.06.2001	24.06.2001	1
7	080047	3	2	20.10.1998	24.10.1998	1
10	136159	3	12	01.04.2001	03.04.2001	0
11	136160	1	6	12.01.1999	12.01.1999	1
17	115705	1	5	15.08.2001	06.09.2001	1
20	080047	3	2	11.10.2001	11.10.2001	1
21	443069	1	2	13.09.2001	14.09.2001	1
22	136160	1	7	18.05.2001	25.05.2001	1

**Рис. 5.13.** Таблица Request после удаления ремонтных заявок исполнителей, выполнивших менее 4 заявок

Для работы вложенного соотнесенного запроса формируется текущая строка-кандидат (т.е. берется первая строка таблицы Request). Внутренний запрос подзапроса находит количество непустых значений дат выполнения ремонтных заявок для кода исполнителя, содержащегося в строке-кандидате (условие

"A.ExecutorCD=B.ExecutorCD"). Запрос DELETE удаляет все строки со значением поля ExecutorCD, входящим во множество значений, формируемых вложенным соотнесенным запросом. В итоге будут удалены все строки таблицы Request, в которых значения поля ExecutorCD равны 2, 4 и 5 (количество выполненных ими заявок 2, 3 и 3 соответственно).

## 5.6. Обновление представлений

Как было отмечено, одной из операций над представлениями является их непосредственное использование с запросами модификации DML: INSERT, UPDATE и DELETE. Такие представления называются *модифицируемыми* (или обновляемыми). Представление можно обновлять, если определяющий его запрос соответствует следующим требованиям:

- должен отсутствовать оператор DISTINCT; т.е. повторяющиеся строки не должны исключаться из таблицы результатов запроса;
- в предложении FROM должна быть задана только одна таблица, которую можно обновлять; т.е. у представления должна быть одна исходная таблица, а пользователь должен иметь соответствующие права доступа к ней. Если исходная таблица сама является представлением, то оно также должно удовлетворять этим условиям;
- каждое имя в списке возвращаемых столбцов должно быть ссылкой на простой столбец; т.е. в этом списке не должны содержаться выражения, вычисляемые столбцы или агрегатные функции;
- предложение WHERE не должно содержать вложенный запрос; т.е. в нем могут присутствовать только простые условия поиска;
- в запросе не должно содержаться предложение GROUP BY или HAVING.

Эти требования базируются на том принципе, что представление разрешается обновлять в том случае, если СУБД может для каждой строки представления найти исходную строку в исходной таблице, а для каждого обновляемого столбца представления – исходный столбец в исходной таблице. Если представление соответствует этим требованиям, то над ним и, как следствие, над исходной таблицей можно выполнять имеющие смысл операции вставки, удаления и обновления. Например, следующие представления являются представлениями только для чтения:

/\* Представление Dailyrequest только для чтения из-за наличия DISTINCT в запросе \*/

```
CREATE VIEW Dailyrequest
AS SELECT DISTINCT AccountCD, RequestCD, IncomingDate,
FailureCD
FROM Request;
```

/\* Представление Summtotal только для чтения из-за наличия агрегатной функции в списке возвращаемых элементов, обращения к двум таблицам и использования предложения GROUP BY \*/

```
CREATE VIEW Summtotal (Abonent, Summa)
AS SELECT Fio, SUM(NachislSum)
```

```
FROM Abonent A, NachislSumma N
WHERE A.AccountCD = N.AccountCD
GROUP BY Fio;
```

/\* Представление Summ\_Abonent только для чтения из-за наличия вложенного запроса в запросе SELECT \*/

```
CREATE VIEW Summ_Abonent
AS SELECT * FROM Abonent WHERE AccountCD IN
(SELECT AccountCD FROM NachislSumma WHERE NachislSum = 46);
```

**Примечание.** Пояснения, стоящие перед запросами CREATE VIEW, представляют собой комментарии SQL и будут подробно рассмотрены далее при изучении процедурного языка.

Рассмотрим примеры работы с обновляемыми представлениями, которые наглядно демонстрируют, как изменяются данные представления при манипулировании данными таблицы, на основе которой оно создано, и как изменяются данные таблицы при манипулировании данными представления.

Пусть требуется создать смешанное представление, которое содержит снимок таблицы Abonent. Вертикальное подмножество должно включать в себя столбцы с фамилиями и номерами лицевых счетов абонентов, а горизонтальное подмножество – строки со значением номера лицевого счёта абонента, больше или равного 200000. Для этого можно использовать следующий запрос:

```
CREATE VIEW Abonent_View (AccountCD, Fio)
AS SELECT AccountCD, Fio FROM Abonent
WHERE AccountCD >= '200000' WITH CHECK OPTION;
```

Результат выполнения запроса

```
SELECT * FROM Abonent_View;
```

представлен на рис. 5.14.

ACCOUNTCD	FIO
443690	ГУЛУПОВА М.И.
443069	СТАРОДУБЦЕВ Е.В.

**Рис. 5.14.** Данные представления Abonent\_View

Это представление является обновляемым по следующим признакам:

- является подмножеством одной таблицы (нет соединения таблиц);
- в списке возвращаемых элементов нет вычисляемых выражений и агрегатных функций;
- используется простое условие поиска (без подзапросов);
- запрос SELECT, на котором базируется представление, не содержит DISTINCT, предложений GROUP BY и HAVING.

Следовательно, к этому представлению могут быть применены запросы INSERT, UPDATE и DELETE. Продемонстрируем это с помощью следующих примеров.

Пусть требуется вставить строку в набор данных, возвращаемый представлением. Это можно выполнить, например, с помощью следующего запроса:

```
INSERT INTO Abonent_View (ACCOUNTCD, Fio)
VALUES ('999999', 'ВАСИЛЬЕВ С.В.);
```

Результат выборки всех данных из представления Abonent\_View представлен на рис. 5.15.

ACCOUNTCD	FIO
443690	ГУЛУПОВА М.И.
443069	СТАРДУБЦЕВ Е.В.
999999	ВАСИЛЬЕВ С.В.

**Рис. 5.15.** Данные представления Abonent\_View после вставки

Данные, содержащиеся в таблице Abonent после применения операции вставки в представление Abonent\_View, приведены на рис. 5.16.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
005488	3	4	1	АКСЕНОВ С.А.	556893
115705	3	1	82	МИЩЕНКО Е.В.	769975
015527	3	1	65	КОНЮХОВ В.С.	761699
443690	7	5	1	ГУЛУПОВА М.И.	214833
136159	7	39	1	СВИРИНА З.А.	350003
443069	4	51	55	СТАРДУБЦЕВ Е.В.	683014
136160	4	9	15	ШМАКОВ С.В.	982222
126112	4	7	11	МАРКОВА В.П.	683301
136169	4	7	13	ДЕНИСОВА Е.К.	680305
080613	8	35	11	ЛУКАШИНА Р.М.	254417
080047	8	39	36	ШУБИНА Т.П.	257842
080270	6	35	6	ТИМОШКИНА Н.Г.	321002
999999	<null>	<null>	<null>	ВАСИЛЬЕВ С.В.	<null>

**Рис. 5.16.** Данные таблицы Abonent

Проверим работу предложения WITH CHECK OPTION, которое должно ограничивать операции DML, применяемые к представлению, таким образом, чтобы они не нарушали условия поиска запроса SELECT, на котором базируется представление. Для этого выполним следующий запрос:

```
INSERT INTO Abonent_View (AccountCD, Fio)
VALUES ('100000', 'СКЛЯРОВ А.А.);
```

Этот запрос на вставку не будет выполнен, т.к. содержит вставку значений, нарушающих условие отбора запроса SELECT, с помощью которого представление было создано. Будет выдано сообщение об ошибке следующего содержания:

"Operation violates CHECK constraint on view or table ABONENT\_VIEW" (нарушение ограничения CHECK представления Abonent\_View).

Следует отметить, что если бы при определении представления Abonent\_View не было предложения WITH CHECK OPTION, то предыдущий запрос не вернул бы ошибку. Строка с номером лицевого счета '100000' была бы вставлена в таблицу Abonent, на которой базируется представление, но в самом представлении ее не было бы видно. Чтобы избежать таких противоречий, нужно включать в определение представления раздел WITH CHECK OPTION. Тогда до реального выполнения операций модификации или вставки строк через представление для каждой строки будет проверяться, что она соответствует условиям представления.

Выполним следующий запрос на удаление:

```
DELETE FROM Abonent_View
WHERE AccountCD = '999999';
```

Данные, содержащиеся в таблице Abonent после удаления строки через представление Abonent\_View, приведены на рис. 5.17.

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	FIO	PHONE
005488	3	4	1	АКСЕНОВ С.А.	556893
115705	3	1	82	МИЩЕНКО Е.В.	769975
015527	3	1	65	КОНЮХОВ В.С.	761699
443690	7	5	1	ТУЛУПОВА М.И.	214833
136159	7	39	1	СВИРИНА З.А.	350003
443069	4	51	55	СТАРОДУБЦЕВ Е.В.	683014
136160	4	9	15	ШМАКОВ С.В.	982222
126112	4	7	11	МАРКОВА В.П.	683301
136169	4	7	13	ДЕНИСОВА Е.К.	680305
080613	8	35	11	ЛУКАШИНА Р.М.	254417
080047	8	39	36	ШУБИНА Т.П.	257842
080270	6	35	6	ТИМОШКИНА Н.Г.	321002

**Рис. 5.17.** Данные таблицы Abonent после удаления

Рассмотрим еще один пример работы с обновляемым представлением.

Пусть необходимо создать представление, которое показывало бы всю информацию об абонентах, кроме информации о номере телефона абонента. При этом должны быть исключены из рассмотрения абоненты с фамилиями, начинающимися с буквы М. Запрос на создание такого представления будет выглядеть следующим образом:

```
CREATE VIEW Abonent_Information
(Code, Street, House, Flat, Name)
AS SELECT AccountCD, StreetCD, HouseNO, FlatNO, Fio
FROM ABONENT
WHERE Fio NOT LIKE 'M%' WITH CHECK OPTION;
```

Данное представление является обновляемым, потому что запрос основан на одной таблице, каждый столбец представления ссылается на простой столбец и предложение WHERE не содержит вложенного запроса. Данные, которые пользователь увидит после выполнения запроса

```
SELECT * FROM Abonent_Information;
```

представлены на рис. 5.18.

CODE	STREET	HOUSE	FLAT	NAME
005488	3	4	1	АКСЕНОВ С.А.
015527	3	1	65	КОНЮХОВ В.С.
443690	7	5	1	ГУЛУПОВА М.И.
136159	7	39	1	СВИРИНА З.А.
443069	4	51	55	СТАРОДУБЦЕВ Е.В.
136160	4	9	15	ШМАКОВ С.В.
136169	4	7	13	ДЕНИСОВА Е.К.
080613	8	35	11	ЛУКАШИНА Р.М.
080047	8	39	36	ШУБИНА Т.П.
080270	6	35	6	ТИМОШКИНА Н.Г.

**Рис. 5.18.** Данные представления Abonent\_Information

Допустим, что был выполнен запрос на изменение данных в таблице Abonent:

```
UPDATE Abonent
SET Fio = 'БУЛГАКОВА Т.П.' WHERE AccountCD = '080047';
```

Также пользователем был выполнен следующий запрос на добавление данных в представление Abonent\_Information:

```
INSERT INTO Abonent_Information
(Code, Street, House, Flat, Name)
VALUES ('123456', 3, 12, 34, 'ТАРАСОВ А.В.);
```

Так как столбец Phone может содержать NULL-значения, то вставка строки в представление проходит успешно. Данные, получаемые в результате выполнения запроса

```
SELECT * FROM Abonent_Information;
```

представлены на рис. 5.19.

CODE	STREET	HOUSE	FLAT	NAME
005488	3	4	1	АКСЕНОВ С.А.
015527	3	1	65	КОНЮХОВ В.С.
443690	7	5	1	ГУЛУПОВА М.И.
136159	7	39	1	СВИРИНА З.А.
443069	4	51	55	СТАРОДУБЦЕВ Е.В.
136160	4	9	15	ШМАКОВ С.В.
136169	4	7	13	ДЕНИСОВА Е.К.
080613	8	35	11	ЛУКАШИНА Р.М.
080047	8	39	36	БУЛГАКОВА Т.П.
080270	6	35	6	ТИМОШКИНА Н.Г.
123456	3	12	34	ТАРАСОВ А.В.

**Рис. 5.19.** Данные представления Abonent\_Information после вставки строки и обновления таблицы Abonent

Пользователь может также выполнять запросы на обновление и удаление данных в представлении. Допустим, были выполнены последовательно два следующих запроса:



```
UPDATE Abonent_Information SET NAME = 'СИМКИНА З.А.'
WHERE CODE = '136159';
DELETE FROM Abonent_Information WHERE CODE = '123456';
```

Данные, получаемые в результате выполнения запроса

```
SELECT * FROM Abonent_Information;
```

представлены на рис. 5.20.

CODE	STREET	HOUSE	FLAT	NAME
005488	3	4	1	АКСЕНОВ С.А.
015527	3	1	65	КОНЮХОВ В.С.
443690	7	5	1	ТУЛУПОВА М.И.
136159	7	39	1	СИМКИНА З.А.
443069	4	51	55	СТАРОДУБЦЕВ Е.В.
136160	4	9	15	ШМАКОВ С.В.
136169	4	7	13	ДЕНИСОВА Е.К.
080613	8	35	11	ЛУКАШИНА Р.М.
080047	8	39	36	БУЛГАКОВА Т.П.
080270	6	35	6	ТИМОШКИНА Н.Г.

**Рис. 5.20.** Данные представления Abonent\_Information после обновления и удаления

Если пользователь попытается вставить или обновить значение поля NAME для абонента с номером лицевого счета '005488', установив фамилию абонента, например, как МОРОЗОВ А.Н. с помощью следующего запроса:

```
UPDATE Abonent_Information SET NAME = 'МОРОЗОВ А.Н.'
WHERE CODE = '005488';,
```

то подобный запрос потерпит неудачу из-за того, что при создании представления было использовано предложение WITH CHECK OPTION. Будет выдано сообщение об ошибке следующего содержания:

"Operation violates CHECK constraint on view or table Abonent\_Information ".

Если пользователь выполнит следующий запрос на удаление:

```
DELETE FROM Abonent_Information WHERE NAME LIKE 'М%';,
```

то в данном случае ошибки не возникнет, но в самой таблице Abonent никаких изменений не произойдет (в данном случае это не является следствием использования предложения WITH CHECK OPTION, а есть следствие применения в запросе представления условия поиска WHERE Fio NOT LIKE 'М%').

Чтобы пояснить результаты действия раздела *WITH CHECK OPTION*, рассмотрим еще один пример. Допустим, что в базе данных созданы два обновляемых представления: Nach\_Information\_20, показывающее информацию о начислениях с суммами, меньшими 20 (идентификатор факта начисления, номер лицевого счета абонента и начисленная сумма), и Nach\_Information\_15, отображающее данные предыдущего представления с суммами более 15. Запросы на создание этих двух представлений будут выглядеть следующим образом:

```
CREATE VIEW Nach_Information_20 (Fact, Code, Summa)
AS SELECT NachislFactCD, AccountCD, NachislSum
FROM NachislSumma WHERE NachislSum < 20
[WITH CHECK OPTION];
```

```
CREATE VIEW Nach_Information_15
AS SELECT Fact, Code, Summa
FROM Nach_Information_20 WHERE Summa > 15
[WITH CHECK OPTION];
```

Данные, получаемые после выполнения запроса  
**SELECT \* FROM Nach\_Information\_20;**,  
 представлены на рис. 5.21.

FACT	CODE	SUMMA
6	136160	18,30
21	080047	19,56
22	080613	10,60
26	080613	12,60
29	136159	8,30
32	443690	17,80
34	126112	15,30
36	080613	12,60
40	015527	18,32
48	136159	8,30
50	136160	18,30

**Рис. 5.21.** Данные представления Nach\_Information\_20

Данные, получаемые после выполнения запроса  
**SELECT \* FROM Nach\_Information\_15;**,  
 представлены на рис. 5.22.

FACT	CODE	SUMMA
6	136160	18,30
21	080047	19,56
32	443690	17,80
34	126112	15,30
40	015527	18,32
50	136160	18,30

**Рис. 5.22.** Данные представления Nach\_Information\_15

В каждом из представлений Nach\_Information\_20 и Nach\_Information\_15 может отсутствовать или присутствовать предложение WITH CHECK OPTION. Рассмотрим, что будет происходить в каждом из возможных случаев при выполнении следующих двух запросов на модификацию строк (будем называть эти запросы U1 и U2 соответственно):

```
UPDATE Nach_Information_15
SET Summa = Summa + 5;
```

```
UPDATE Nach_Information_15
SET Summa = Summa - 5;.
```

**Случай 1.** Ни в одном из представлений не содержится раздел WITH CHECK OPTION.

Допустим, был выполнен запрос U1 на увеличение начисленных сумм. После выполнения запроса U1 представление Nach\_Information\_15 оказывается пустым. Его строки перестают удовлетворять условию представления Nach\_Information\_20 и исчезают из него. Данные представления Nach\_Information\_20 после выполнения запроса U1 показаны на рис. 5.23.

FACT	CODE	SUMMA
22	080613	10,60
26	080613	12,60
29	136159	8,30
36	080613	12,60
48	136159	8,30

**Рис. 5.23.** Данные представления Nach\_Information\_20 после выполнения запроса U1

Этот результат может быть неожиданным для пользователей базы данных, которым известно, что условие представления Nach\_Information\_15 имеет вид  $Summa > 15$  и соблюдение этого условия должно сохраняться при увеличении размера начисления. В то же время в таблице NachislSumma обновляются значения полей NachislSum для 6 записей (с идентификаторами факта начисления 6, 21, 32, 34, 40, 50).

После выполнения запроса U2 представление Nach\_Information\_15 также оказывается пустым. Данные представления Nach\_Information\_20 после выполнения запроса U2 показаны на рис. 5.24.

FACT	CODE	SUMMA
6	136160	13,30
21	080047	14,56
22	080613	10,60
26	080613	12,60
29	136159	8,30
32	443690	12,80
34	126112	10,30
36	080613	12,60
40	015527	13,32
48	136159	8,30
50	136160	13,30

**Рис. 5.24.** Данные представления Nach\_Information\_20 после выполнения запроса U2

Возможно, результат будет достаточно естественным для пользователей представления Nach\_Information\_15, которым известно условие представления, но те, кто работает с представлением Nach\_Information\_20, обнаружат в теле результирующей таблицы 6 строк с измененными значениями полей Summa (с идентификаторами факта начисления 6, 21, 32, 34, 40, 50).

В таблице NachSumma также будут изменены 6 записей.

**Случай 2.** В определении представления Nach\_Information\_20 содержится раздел WITH CHECK OPTION, а в определении Nach\_Information\_15 раздел WITH CHECK OPTION отсутствует.

В этом случае запрос U1 будет отвергнут системой (поскольку его выполнение нарушает условие представления Nach\_Information\_20). Будет выдано следующее сообщение: "Operation violates CHECK constraint on view or table NACH\_INFORMATION\_20".

Но заметим, что такое поведение системы будет совершенно неожиданным и непонятным для тех пользователей базы данных, которым известно только определение представления Nach\_Information\_15, поскольку запрос U1 явно не может нарушить видимое ими ограничение.

С другой стороны, запрос U2 будет выполнен и по-прежнему приведет к опустошению представления Nach\_Information\_15. Те, кто работает с представлением Nach\_Information\_20, обнаружат в теле результирующей таблицы 6 строк с измененными значениями полей NachSum, как и в предыдущем случае (см. рис. 5.24).

**Случай 3.** В определении представления Nach\_Information\_20 раздел WITH CHECK OPTION отсутствует, а в определении Nach\_Information\_15 содержится раздел WITH CHECK OPTION.

Запрос U1 (увеличение размера начисления) будет успешно выполнен, поскольку он не противоречит локальным ограничениям представления NACH\_INFORMATION\_15. После выполнения запроса U1 представление Nach\_Information\_15 оказывается пустым. Его строки перестают удовлетворять условию представления Nach\_Information\_20 и исчезают из него. Данные представления Nach\_Information\_20 будут совпадать с данными, показанными на рис. 5.23 (случай 1).

В таблице NachSumma также будут изменены 6 записей.

Запрос U2 не будет работать (его выполнение не будет допущено условием проверки представления NACH\_INFORMATION\_15).

**Случай 4.** В определении представлений Nach\_Information\_20 и Nach\_Information\_15 содержится раздел WITH CHECK OPTION.

В этом случае запрос U1 не будет выполнен, так как его выполнение нарушает условие проверки представления NACH\_INFORMATION\_20, а запрос U2 не будет выполнен, так как его выполнение нарушает условие проверки представления NACH\_INFORMATION\_15.

Только в этом случае операции обновления будут выполняться корректно. Очевидный вывод из приведенного анализа заключается в том, что единственным способом обеспечить корректность выполнения операций обновления через представления (допускающие операции обновления) является включение в определение каждого имеющегося в БД представления предложения WITH CHECK OPTION.

## Контрольные вопросы

1. Какие запросы языка DML используются в SQL СУБД Firebird?
2. Как осуществляется добавление новых данных в таблицу с помощью однострочного запроса?
3. Как добавить в таблицу строки из другой таблицы?
4. Какие существуют ограничения на подзапрос, используемый в многострочном запросе INSERT?
5. Как осуществляется простое обновление данных в таблице БД?
6. Как могут использоваться подзапросы при обновлении данных?
7. Как построить запрос на обобщенное добавление и обновление данных средствами языка SQL?
8. Как выполнить обновление или вставку строк в одной таблице на основании данных другой таблицы?
9. Как выполнить простое удаление данных из таблицы БД?
10. Каким образом работает запрос на удаление данных с подзапросом?
11. В каком случае к представлению можно применять запросы DML?
12. Какие проблемы могут возникнуть, если при создании представления не использовалось предложение WITH CHECK OPTION?

## 6. Процедурный язык

Эта глава посвящена изучению процедурного языка SQL, который предполагает расширение стандартного набора запросов, изученных в предыдущих главах, средой программирования.

В СУБД Firebird существует возможность создания независимых исполняемых модулей, представленных разработчиками в виде исходных кодов. Такие коды выполняются полностью на сервере, возвращая при необходимости клиентскому приложению значения, полученные в результате выполнения. Язык, который предоставляет такую возможность для сервера Firebird, называется PSQL (процедурный SQL). Это простой, но мощный набор расширений языка SQL, которых нет в стандарте SQL2, но которые включены в стандарт языка SQL3. Такими расширениями являются триггеры и хранимые процедуры.

В настоящей главе, прежде всего, рассмотрен ряд предварительных вопросов, относящихся к процедурному языку, без знания которых невозможно создавать и использовать триггеры и хранимые процедуры. Далее приведены

особенности создания и работы с хранимыми процедурами и триггерами. Здесь же отдельно рассмотрен оператор EXECUTE BLOCK, который предоставляет возможность выполнения кода на PSQL без оформления именованной ХП. Такой фрагмент кода на PSQL подобен хранимой процедуре (может иметь входные и выходные параметры, результат работы идентичен результату ХП).

## 6.1. Основы разработки модулей на PSQL

Процедурный язык Firebird – это язык программирования, используемый для написания хранимых процедур и триггеров в СУБД Firebird. Он может включать стандартные запросы SQL, а также расширения языка SQL.

Расширения языка PSQL в Firebird включают следующие языковые элементы:

- объявление и инициализация локальных переменных, оператор присваивания;
- условные операторы (оператор ветвления и оператор цикла);
- явный и неявный курсоры для выполнения цикла при просмотре строк;
- генератор последовательности целых значений;
- генерация сообщения об исключительной ситуации и прерывание выполнения кода триггера или хранимой процедуры;
- оператор EXECUTE STATEMENT для выполнения запросов DML и DDL в модуле.

Существует следующий ряд ограничений языка для кодов в модулях PSQL:

- запросы языка определения данных (DDL) не разрешены в PSQL (передача DDL-запроса возможна только с помощью EXECUTE STATEMENT);
- команды управления транзакциями недопустимы в PSQL, потому что хранимые процедуры и триггеры всегда выполняются в контексте существующей клиентской транзакции, а Firebird не поддерживает вложенные транзакции.

### 6.1.1. Переменные

В модулях на PSQL может быть использовано четыре типа переменных, причем возможность использования конкретной переменной определяется тем, является ли модуль хранимой процедурой или триггером. Различают локальные переменные, контекстные переменные, входные параметры и выходные параметры.

Локальные переменные используются для хранения значений как в ХП, так и в триггерах. Объявление локальной переменной в теле триггера или хранимой процедуры осуществляется с помощью следующего оператора:

```
DECLARE [VARIABLE] имя_локальной_переменной <тип_переменной> [{ = |  
DEFAULT} <значение>];
```

где

<тип\_переменной> ::= {<тип\_данных> | имя\_домена | TYPE OF имя\_домена}.

Следует обратить внимание на то, что для каждой переменной используется свой оператор `DECLARE VARIABLE`, который заканчивается точкой с запятой.

Правило составления имен локальных переменных такое же, как и для любых других идентификаторов языка SQL (см. п. 2.7).

В качестве типа данных можно указывать один из системных типов (см. п. 2.9), ранее созданный домен или конструкцию `TYPE OF <имя_домена>`, которая извлекает только тип из существующего домена (т.е. ограничения и значения по умолчанию, определенные для данного домена, не учитываются).

Например, чтобы объявить переменную `x` целого типа, необходимо использовать следующий оператор:

```
DECLARE VARIABLE x INTEGER;
```

Объявить переменную `y` на домене `BOOLEAN` можно следующим образом:

```
DECLARE VARIABLE y BOOLEAN;
```

Все локальные переменные должны быть объявлены до первого выполняемого блока триггера (процедуры).

Объявленным переменным в дальнейшем могут присваиваться различные значения с помощью оператора присваивания, имеющего следующий синтаксис:

```
имя_локальной_переменной = <выражение>;
```

Переменным должны присваиваться значения того типа данных, с каким они были объявлены.

Допускается при объявлении локальной переменной сразу же ее инициализировать, например следующим образом:

```
DECLARE VARIABLE x INTEGER = 123;
```

или

```
DECLARE VARIABLE x INTEGER DEFAULT 123;.
```

СУБД делает доступным множество переменных, поддерживаемых в контексте текущего соединения клиента и его деятельности. Эти контекстные переменные доступны для использования в `PSQL`. Большинство контекстных переменных процедурного языка используются только в триггерах. Существует только одна контекстная переменная `PSQL`, которую можно использовать как в триггерах, так и в ХП. Это переменная `ROW_COUNT`, возвращающая количество строк, полученных выполненным запросом `DML` или `SELECT`.

В триггерах для реализации отслеживания целостности данных используются контекстные переменные `OLD.столбец` и `NEW.столбец`. Эти переменные хранят старые и новые значения указанного столбца таблицы, когда выполняется запрос на модификацию данных.

Также в триггерах могут использоваться логические контекстные переменные `UPDATING`, `INSERTING` и `DELETING` для задания определенных действий в зависимости от типа события `DML`, для которого срабатывает триггер.

Входные параметры недоступны для использования в триггерах. Они используются для передачи значений хранимым процедурам из клиентских приложений или других хранимых процедур.

Выходные параметры также недоступны для использования в триггерах. Они используются для возвращения значений из ХП вызвавшим их объектам.

Говоря о переменных процедурного языка, следует особо отметить то, что в запросах SQL локальные переменные, входные и выходные параметры должны использоваться с двоеточием перед именем переменной, чтобы указать, что это переменные, а не имена столбцов таблиц. В остальных случаях (например, при присваивании значения переменной, в логических условиях условных операторов) не требуется наличия двоеточия перед именем переменной. Двоеточие перед контекстными переменными не ставится никогда.

Примеры практического использования всех четырех типов переменных в модулях на PSQL приведены при изучении ХП и триггеров.

## 6.1.2. Условные операторы

При программировании на PSQL существует возможность использовать следующие типы условных структур:

- ветвление, управляемое оператором IF...THEN...ELSE;
- циклическое выполнение группы операторов, пока условие WHILE не станет ложным.

### 6.1.2.1. Оператор ветвления IF

Оператор ветвления IF имеет следующий формат:

```
IF (логическое_условие) THEN <группа_операторов1>  
[ELSE <группа_операторов2>],
```

где логическое\_условие – любой предикат, который должен быть истинным, чтобы выполнялась <группа\_операторов>, следующая за THEN;

```
<группа_операторов> ::= { <блок_операторов>  
                        | DML_оператор  
                        | оператор_процедурного_языка_Firebird };
```

```
<блок_операторов> ::=  
BEGIN  
  [<группа_операторов> ...]  
END.
```

Блок операторов – это группа операторов SQL, рассматриваемых как единое целое. Блок операторов начинается с зарезервированного слова BEGIN и заканчивается зарезервированным словом END. При этом блок операторов



может содержать вложенный блок операторов, заключенный в операторные скобки BEGIN END, и т.д. Для вложенных блоков после зарезервированного слова END разделительная точка с запятой не ставится. Следует отметить, что между BEGIN...END может не быть ни одного оператора, т.е. допустим "пустой" блок операторов.

Следует отметить, что если <группа\_операторов> представляет собой один единственный DML\_оператор или оператор\_процедурного\_языка\_Firebird, то заключать данный оператор в скобки BEGIN ... END не обязательно.

Следующий фрагмент кода иллюстрирует использование оператора IF в теле ХП в предположении, что Street\_Cod, House\_Nom, Flat\_Nom были ранее объявлены как локальные переменные или входные параметры, а Line – как локальная переменная или выходной параметр:

```
...
IF (House_Nom IS NOT NULL AND Flat_Nom IS NOT NULL) THEN
    Line = 'Код улицы - ' || Street_Cod || ', д.' || House_Nom || ', кв.'
        || Flat_Nom;
ELSE
    Line = Street_Cod;
...

```

Как следует из примера, в конце каждого оператора ставится точка с запятой, в том числе и перед ELSE.

### 6.1.2.2. Оператор WHILE

Для организации цикла с предусловием можно использовать оператор WHILE, который имеет следующий формат:

```
WHILE (логическое_условие) DO
    <группа_операторов>.

```

Следующий фрагмент кода иллюстрирует использование в ХП оператора WHILE в предположении, что Digit и Result были ранее объявлены как локальные переменные или параметры:

```
...
Result = CAST (Digit as VARCHAR(8));
WHILE (CHAR_LENGTH(Result)<8) DO Result = '0' || Result;
...

```

В этом фрагменте переданное целое число (Digit) преобразовывается в строку из 8 символов (Result), а затем дополняется нулями слева до получения нужного количества символов.

### 6.1.3. Курсоры в PSQL

При написании хранимых процедур и триггеров очень часто возникает необходимость выбрать информацию из БД перед выполнением каких-либо

других действий. Для таких целей в модулях на PSQL допускается использовать запрос на выборку следующего вида:

```
SELECT ... INTO <список_переменных>;
```

Однако запрос такого вида можно использовать только в случае, если он возвращает одно единственное значение (скалярный запрос) или только одну строку из одной или нескольких таблиц БД.

В большинстве случаев запрос к реляционной базе данных возвращает несколько записей данных, но приложение за один раз обрабатывает лишь одну запись. При модификации, удалении или добавлении данных с помощью запросов DML действия выполняются также над отдельной записью (строкой) таблицы. В этой ситуации на первый план выступает концепция курсора. Под курсором понимается получаемый при выполнении запроса результирующий набор и связанный с ним указатель текущей записи. Обычно курсоры используются для выбора из базы данных некоторого подмножества хранимой в ней информации. В каждый момент времени прикладной программой может быть обработана одна строка курсора. После позиционирования курсора над этой строкой можно выполнять различные действия.

Курсоры часто применяются в запросах SQL, встроенных в прикладные программы, написанные на языках процедурного типа.

PSQL СУБД Firebird поддерживает два типа курсоров: явные и неявные. Реализация неявного курсора представлена следующей конструкцией:

```
FOR SELECT ... INTO ... DO ... .
```

Она полностью реализует синтаксис цикла и предоставляет возможность последовательной построчной обработки набора данных курсора в цикле FOR.

Явный курсор, также называемый изменяемым или именованным курсором, представляет собой объект, который разработчик может явно объявлять, открывать и закрывать (в отличие от неявного курсора, не требующего объявления, открытия и закрытия). Для управления явным курсором используются команды DECLARE CURSOR, OPEN, FETCH и CLOSE. Рассмотрим более подробно особенности этих двух реализаций курсора.

### 6.1.3.1. Неявный курсор

Как известно, запрос SELECT возвращает таблицу результатов запроса, но приложения не всегда могут эффективно работать с результирующим набором. Возникает проблема доступа к каждой строке этой таблицы, которая может быть решена при использовании неявного курсора.

В процедурном языке Firebird цикл для построчной обработки набора данных неявного курсора организуется с помощью следующей конструкции:

```
FOR
SELECT [DISTINCT | ALL]
  { <возвращаемый_элемент1> [, <возвращаемый_элемент2> ] ... }
FROM базовая_таблица1 [, базовая_таблица2 ] ...
```

```

[WHERE <условие_поиска>]
[GROUP BY <элемент_группировки1> [, <элемент_группировки2>]... ]
[HAVING <условие_поиска>]
[ORDER BY <элемент_сортировки1> [, <элемент_сортировки2>]...]
INTO :<имя_переменной1> [, :<имя_переменной2> ...]
[AS CURSOR имя_курсора]
DO <группа_операторов>,
где
<имя_переменной>:: = {имя_локальной_переменной
                       | входной_параметр | выходной_параметр}.

```

Как следует из приведенного синтаксиса, набор данных курсора определяется запросом SELECT, который может соединять данные из нескольких таблиц, содержать условия поиска, сортировку и т.д.

Для обработки всех строк, возвращаемых запросом SELECT, организуется цикл. При этом в запросе SELECT используется предложение INTO. Оно определяет, что значения возвращаемых элементов должны быть присвоены переменным, определенным в теле ХП, триггера или блока. Предложение INTO в запросе SELECT должно быть последним, если он имеет другие предложения (например, WHERE).

Каждый раз, когда обрабатывается очередная строка в цикле, значения возвращаемых элементов присваиваются переменным, указанным в предложении INTO. При этом <группа\_операторов> выполняется один раз для каждой строки, сформированной запросом SELECT. Цикл повторяется, пока не закончатся записи в наборе данных курсора.

Например, в теле ХП использование неявного курсора может выглядеть следующим образом:

```

DECLARE LAccountCD VARCHAR(6);
DECLARE LFiо VARCHAR(20);
DECLARE LPayDate DATE;
DECLARE LPaySum NUMERIC(15,2);
BEGIN
FOR SELECT A. AccountCD, A.Fio, P. PayDate, P. PaySum
FROM Abonent A, PaySumma P
WHERE A. AccountCD = P. AccountCD AND P. PaySum > 70
INTO :LAccountCD, :LFio, :LPayDate, :LPaySum
DO
SUSPEND;
END.

```

Следует отметить важную особенность работы с курсорами. Они позволяют обеспечить обновление или удаление строки, на которой в данный момент установлен курсор, с помощью конструкции WHERE CURRENT OF в запросе UPDATE или DELETE. В этом случае <группа\_операторов> при обновлении строки, на которой установлен курсор, должна выглядеть следующим образом:

```

UPDATE базовая_таблица
SET ... WHERE CURRENT OF имя_курсора;.

```

При удалении строки, на которой установлен курсор, <группа\_операторов> должна выглядеть следующим образом:

```
DELETE FROM базовая_таблица  
WHERE CURRENT OF имя_курсора;.
```

За одну операцию обновления могут быть изменены несколько столбцов текущей строки курсора, но все они должны принадлежать одной таблице. Если выполняется удаление, то будет удалена строка, установленная текущей в курсоре.

### 6.1.3.2. Явный курсор

Работа с явным курсором предполагает его определение, связывание с ним запроса SELECT, открытие, выборку данных из курсора и его закрытие.

Для определения явного курсора и связывания с ним запроса SELECT используется следующий оператор:

```
DECLARE [VARIABLE] имя_курсора CURSOR FOR (<запрос_select>);.
```

Для открытия курсора используется следующий оператор:

```
OPEN имя_курсора;.
```

Для выборки данных из курсора используется следующий оператор:

```
FETCH имя_курсора INTO :<имя_переменной1> [, :<имя_переменной2> ...];.
```

Для закрытия курсора используется следующий оператор:

```
CLOSE имя_курсора;.
```

**Примечание.** Команды OPEN, FETCH и CLOSE, используемые для работы с явным курсором, нельзя применять для работы с неявным курсором.

Объявление курсора должно помещаться в начале триггера, ХП или неименованного выполняемого блока (заданного с помощью EXECUTE BLOCK), подобно объявлению обычных локальных переменных.

Например, в теле ХП объявление и использование явного курсора может выглядеть следующим образом:

```
DECLARE Aname CHAR(31);  
DECLARE c CURSOR FOR (SELECT Fio FROM Abonent);  
BEGIN  
  OPEN c;  
  WHILE (1 = 1) DO  
  BEGIN  
    FETCH c INTO :Aname;  
    IF (ROW_COUNT = 0) THEN  
      LEAVE;  
    SUSPEND;  
  END  
  CLOSE c;  
END.
```

В данном примере показано, как может быть организована выборка данных с использованием явного курсора. Организуется цикл WHILE, внутри которого

контекстная переменная ROW\_COUNT проверяет, возвращает ли последняя выборка (FETCH) из явного курсора строку данных. Если нет, то происходит выход из цикла WHILE с помощью оператора LEAVE и закрытие курсора.

Следует помнить, что имена курсоров должны быть уникальными в контексте того модуля, где курсоры используются. Т.е. каждый явный и неявный курсор (если для него объявлено имя с помощью конструкции AS CURSOR имя\_курсора) должны иметь различные имена. Однако допускается совпадение имени курсора с именем любой переменной, используемой в том же самом модуле.

Явный курсор, как и неявный, допускает позиционированные обновления и удаления.

Попытки открыть курсор, который уже открыт, так же как и попытки выбрать информацию из уже закрытого курсора или закрыть его вновь, будут неудачны. Все курсоры, которые явно не были закрыты, будут закрыты автоматически при выходе из текущего PSQL блока, процедуры или триггера.

#### **6.1.4. SQL сценарии**

Как правило, создание триггеров и хранимых процедур осуществляется с использованием SQL сценария.

SQL сценарий (файл-сценарий, скрипт) – это текстовый файл, содержащий запросы и операторы SQL и обычно имеющий расширение sql. Все запросы, команды и операторы SQL сценария выполняются последовательно в том порядке, в котором они следуют в скрипте, и должны быть отделены друг от друга разделителем (точкой с запятой или тем, который задан с помощью оператора SET TERM).

В файле SQL сценария рекомендуется использовать разные разделители для разделения внутренних операторов ХП или триггера от внешних запросов (команд).

Новый внешний разделитель определяется оператором SET TERM в следующем формате:

```
SET TERM новый_разделитель старый_разделитель.
```

Например, чтобы определить в качестве разделителя два восклицательных знака (!! ) вместо обычного разделителя (;), необходимо использовать следующий оператор:

```
SET TERM !! ;.
```

После этого точка с запятой будет использоваться для разделения внутренних операторов ХП или триггера, а два восклицательных знака – для отделения ХП, триггеров и других запросов (команд) друг от друга в файле SQL сценария. После использования нового разделителя восстановить старый разделитель можно следующей командой:

```
SET TERM ; !!
```

SQL сценарий может содержать запросы на создание всех объектов БД, т.е. фактически являться моделью БД.

Запуск на выполнение SQL сценария в СУБД Firebird производится выбором пункта «Выполнить скрипт» (<F9>) меню редактора скриптов (<Ctrl+F12>) утилиты IBExpert. При выполнении скрипта происходит компиляция хранимых процедур и триггеров, включенных в текст данного SQL-скрипта.

В общем случае скрипт может содержать запросы модификации БД (язык DDL), запросы изменения данных (язык DML) и команды фиксации/отмены транзакций. Запросы выборки данных в скрипте использовать нельзя (кроме передачи их в строке с помощью оператора EXECUTE STATEMENT, который будет подробно рассмотрен позднее, или для задания набора данных курсора).

Текстовый файл скрипта также может содержать команду на создание БД, которая имеет следующий синтаксис:

```
CREATE DATABASE 'filespec'  
[USER 'username' [PASSWORD 'password']]  
[PAGE_SIZE [=] int ]  
[DEFAULT CHARACTER SET charset];,
```

где

filespec – спецификация файла новой БД;

[USER 'username'] – задает имя пользователя;

[PASSWORD 'password'] – задает пароль;

[PAGE\_SIZE [=] int] – устанавливает размер в байтах страниц БД [допустимо 1024, 2048, 4096 (по умолчанию), 8192 и 16384];

[DEFAULT CHARACTER SET charset] – устанавливает набор символов с именем charset для БД, используемый по умолчанию. Если опущено, то в качестве набора символов по умолчанию принимается NONE.

Задание *кодовой таблицы* (набора символов), используемой по умолчанию для данных, хранимых в базе, имеет существенное значение при создании БД. Для хранения данных на русском языке пригодны два следующих варианта.

#### 1. Создание БД без задания кодовой таблицы.

В этом случае символьные данные хранятся в базе в том виде, как они были загружены, без каких-либо предварительных преобразований. Сортировка данных осуществляется в порядке возрастания кодов хранимых символов.

Команда на создание БД без задания кодовой таблицы имеет следующий синтаксис:

```
CREATE DATABASE 'filespec'  
[USER 'username' [PASSWORD 'password']]  
[PAGE_SIZE [=] int ]  
[DEFAULT CHARACTER SET NONE];.
```

Например, для создания БД без задания кодовой таблицы можно использовать следующую команду:

```
CREATE DATABASE 'c:\sqllab.fdb' -- спецификация файла БД  
USER 'SYSDBA' PASSWORD 'masterkey'  
PAGE_SIZE 4096  
DEFAULT CHARACTER SET NONE;
```

#### 2. Создание БД с кодовой таблицей WIN1251.

В этом случае команда на создание БД имеет следующий синтаксис:

```
CREATE DATABASE 'filespec'  
[USER 'username' [PASSWORD 'password']]  
[PAGE_SIZE [=] int ]  
[DEFAULT CHARACTER SET WIN1251];.
```

Важной характеристикой БД, помимо размера страницы и набора символов, является ее диалект. Создание новой БД в Firebird возможно в диалекте 1 или диалекте 3, однако рекомендуется создавать новые базы в диалекте 3, так как он предоставляет наиболее полный набор возможностей Firebird. Учебная база данных, используемая в данном пособии, создана на диалекте 3.

Чтобы задать диалект создаваемой БД в скрипте, следует перед командой CREATE DATABASE выполнить следующую команду:

```
SET SQL DIALECT номер_диалекта;,  
где номер_диалекта равен 1 или 3.
```

**Примечание.** При создании БД с использованием IBExpert диалект, также как размер страницы и набор символов, выбирается из выпадающего списка в окне «Создание базы данных».

Каждый скрипт должен содержать подключение к БД, которое производится с помощью команды CONNECT, имеющей следующий формат:

```
CONNECT 'filespec'  
[USER 'username'] [PASSWORD 'password'] [ROLE 'rolename'];.
```

Например, для подключения к созданной базе данных с именем sgllab.fdb следует использовать следующую команду:

```
CONNECT 'c:\sqllab.fdb' USER 'SYSDBA'  
PASSWORD 'masterkey';.
```

Для автоматического помещения данной команды в начало создаваемого скрипта достаточно в окне редактора скриптов выбрать в меню «Скрипт» пункт «Добавить оператор CONNECT в начало скрипта». В появившемся окне нужно выбрать БД, для которой будет использоваться скрипт.

Скрипт может содержать команду DISCONNECT для принудительного разрыва соединения с БД после выполнения необходимых операций. Команда DISCONNECT имеет следующий синтаксис:

```
DISCONNECT 'filespec';.
```

Например, для отсоединения от подключенной учебной БД с именем sgllab.fdb можно использовать следующую команду:

```
DISCONNECT 'c:\sqllab.fdb';.
```

В файле SQL сценария можно использовать комментарии, которые предоставляют возможность приводить объяснения и любую другую полезную информацию. На выполнение команд SQL комментарии не влияют.

Различают следующие типы комментариев:

- *блочный*, который содержит одну или несколько строк комментария. В основном блочный комментарий располагается на отдельной строке (или строках), но может располагаться и в той же строке, что и оператор SQL.

Такой комментарий отделяется от основного текста сценария следующим образом:

```
/* comment_text */
```

- *однотрочный*, который располагается в конце строки кода в скрипте, DDL или DML-запросе, хранимой процедуре, триггере. Отделяется от основного текста сценария следующим образом:

```
-- comment_text
```

Однотрочный комментарий часто используется для комментирования нежелательных частей утверждений, например в следующем виде:

```
... WHERE C1 = 9 OR C2 = 99 -- OR C3 = 999 ...
```

В данном случае часть условия поиска (OR COL3 = 999) закомментирована. При использовании данного вида комментария игнорируется текст до следующей строки.

### **Примечания.**

1. Рекомендуется создавать комментарии, используя латинские символы, т.к. извлечение комментариев с кириллицей может вызывать ошибки.

2. Не допускается смешивание однотрочного комментария с блочным комментарием.

Пример SQL-сценария по созданию учебной базы данных и заполнению ее данными приведен в приложении Б.

### **6.1.5. Генераторы**

В PSQL существует очень важный с точки зрения практического использования БД механизм порождения уникальных значений, которые могут применяться, например, для уникальной идентификации строк таблиц – генератор последовательности.

Генератор последовательности – это специальный объект БД для получения целочисленных значений, следующих с определенным шагом. В БД Firebird каждый генератор имеет уникальное имя и текущее значение. Для создания генератора используется следующий запрос:

```
CREATE { SEQUENCE | GENERATOR } имя_генератора;
```

При выполнении такого запроса происходит два следующих действия:

- на специальной странице БД отводится 4 байта для хранения значения генератора;
- в системной таблице RDB\$GENERATORS заводится запись, куда помещаются <имя\_генератора> (поле RDB\$GENERATOR\_NAME), его номер (поле RDB\$GENERATOR\_ID) и признак того, что генератор создан пользователем (значение в поле RDB\$SYSTEM\_FLAG равно нулю).

Следует отметить, что по стандарту SQL для обозначения генератора последовательности используется синтаксический термин SEQUENCE, в то время как GENERATOR – это синтаксический термин InterBase. Для



соответствия SQL-стандарту для работы с генераторами последовательностей рекомендуется использовать именно термин SEQUENCE.

По умолчанию генератор создается с текущим значением ноль. Для установки определенного (текущего) значения генератора <целое\_значение> можно использовать команду следующего формата:

```
SET GENERATOR <имя_генератора> TO <целое_значение>;
```

Однако для соответствия SQL-стандарту предпочтительнее использование следующего запроса:

```
ALTER SEQUENCE имя_генератора RESTART WITH целое_значение;
```

При этом целое\_значение – это значение типа BIGINT, которое должно лежать в пределах от  $-2^{63}$  до  $2^{63}-1$ .

Например, генератор с именем Executor\_ID, который может использоваться для автоматической генерации очередного значения первичного ключа таблицы Executor учебной БД, можно определить с помощью следующего запроса:

```
CREATE SEQUENCE Executor_ID;
```

Установка начального значения равным 5 для созданного генератора Executor\_ID может быть выполнена следующим образом:

```
ALTER SEQUENCE Executor_ID RESTART WITH 5;
```

Создание генератора и установка начального значения могут быть произведены при выполнении описанных выше запросов в SQL-редакторе утилиты IBExpert или при использовании этих запросов в тексте SQL сценария.

После создания генератора получение его очередного (следующего) значения производится вызовом функции GEN\_ID или функции NEXT VALUE FOR. Функция GEN\_ID имеет следующий формат:

```
GEN_ID (имя_генератора, шаг),
```

где шаг представляет собой целое значение (если оно равно нулю, то будет получено текущее значение генератора).

Формат функции NEXT VALUE FOR имеет следующий вид:

```
NEXT VALUE FOR имя_генератора;
```

Значение шага при использовании данной функции всегда равно 1. Если нет необходимости использовать приращение шага, отличное от 1, рекомендуется пользоваться функцией NEXT VALUE FOR, так как именно она соответствует наиболее позднему стандарту SQL. Приращение значения генератора, не равное 1, может быть задано только через функцию GEN\_ID.

Функции вызова генератора могут использоваться в следующих случаях:

- для получения очередного значения генератора и присвоения его локальной переменной в теле триггера или хранимой процедуры;
- в списке возвращаемых элементов запроса SELECT;
- при добавлении и изменении строк (INSERT- и UPDATE-запросы DML).

Например, для созданного генератора Executor\_ID получение очередного значения и присвоение его локальной переменной Executor\_Number (в теле триггера или хранимой процедуры) имеют следующий вид:

```
Executor_Number = GEN_ID(Executor_ID, 1);
```

Функции получения значения генератора могут использоваться в списке возвращаемых элементов запроса SELECT. В таком случае в предложении

FROM запроса SELECT указывается системная таблица Firebird RDB\$DATABASE, которая всегда содержит только одну строку со служебной информацией о базе данных. Данная таблица используется для запросов, которые возвращают одно вычисляемое значение или контекстную переменную [18].

Например, для получения текущего значения генератора Executor\_ID в списке возвращаемых элементов запроса SELECT можно использовать следующий запрос:

```
SELECT GEN_ID (Executor_ID,0) FROM RDB$DATABASE;.
```

Результат выполнения этого запроса представлен на рис. 6.1.

GEN_ID
5

**Рис. 6.1.** Результат получения текущего значения генератора

Для получения следующего значения созданного генератора Executor\_ID с помощью функции NEXT VALUE FOR нужно выполнить следующий запрос:

```
SELECT NEXT VALUE FOR Executor_ID FROM RDB$DATABASE;.
```

В результате выполнения этого запроса будет выдано значение, равное 6.

Примером использования генератора последовательности в DML-запросе на вставку строки может служить следующий запрос:

```
INSERT INTO Executor  
VALUES (NEXT VALUE FOR Executor_ID, 'Иванов А.А.);.
```

Следует отметить, что вызов функции вида

```
GEN_ID (<имя_генератора>, 0)
```

никогда не должен использоваться в запросах DML на добавление или изменение данных.

Созданный и неиспользуемый генератор можно удалить путем выполнения следующего запроса:

```
DROP { SEQUENCE | GENERATOR } имя_генератора;.
```

### 6.1.6. Исключительные ситуации

Исключительные ситуации (exception) – это сообщения, которые генерируются, когда появляется ошибка в теле хранимой процедуры или триггера.

В СУБД Firebird существуют предварительно определенные исключения с ассоциированными с ними текстами сообщений. Такие сообщения выдаются пользователю, например, при попытке удалить объект, который находится в использовании, при попытке вставить некорректные данные в столбец, на который наложено ограничение, и т.д. Но, кроме этого, существует возможность создания пользовательских исключений.

Для определения нового пользовательского исключения используется запрос CREATE EXCEPTION, имеющий следующий формат:

```
CREATE EXCEPTION имя_исключения <сообщение>;
```

где <сообщение> - строка текста длиной до 78 символов, заключенная в апострофы.

Например, чтобы создать исключительную ситуацию с именем ExcErr и сообщением 'Ошибка', необходимо использовать следующий запрос:

```
CREATE EXCEPTION ExcErr 'Ошибка';
```

Для изменения текста сообщения служит запрос ALTER EXCEPTION, например:

```
ALTER EXCEPTION ExcErr 'Ошибка преобразования';
```

Если требуется заново создать исключение со старым именем, то используется запрос RECREATE EXCEPTION. Если исключение не существует перед использованием этого запроса, то его использование эквивалентно выполнению запроса CREATE EXCEPTION. Если исключение уже существует, то запрос RECREATE EXCEPTION удаляет его (если оно не используется другими объектами) и создает полностью новый объект, например в следующем случае:

```
RECREATE EXCEPTION ExcErr 'Ошибка преобразования данных';
```

Также может использоваться запрос CREATE OR ALTER EXCEPTION, который создает исключение, если оно не существует, или изменяет определение существующего исключения (даже если оно используется другими объектами БД).

Для удаления исключительной ситуации используется запрос DROP EXCEPTION, например:

```
DROP EXCEPTION ExcErr;
```

Следует отметить, что создать, пересоздать, изменить или удалить исключительную ситуацию можно как при выполнении соответствующего запроса в SQL-редакторе IBExpert, так и в тексте SQL сценария.

Чтобы активизировать в теле триггера или хранимой процедуры генерацию сообщения об исключительной ситуации и тем самым прервать выполнение триггера или хранимой процедуры, необходимо использовать следующую команду:

```
EXCEPTION имя_исключения; .
```

При этом действия, которые были произведены в триггере или хранимой процедуре над данными в текущей транзакции, отменяются и выполнение ХП или триггера прекращается.

Следует отметить, что можно заранее не создавать исключение, а генерировать исключительную ситуацию с сообщением, задаваемым во время выполнения. В таком случае активизация сообщения имеет следующий формат:

```
EXCEPTION имя_исключения <сообщение>;
```

При создании учебной базы данных определены три исключения: Ins\_Restrict, Del\_Restrict и Upd\_Restrict (приложение Б). Примеры использования этих исключений, а также примеры определения и использования других исключений будут рассмотрены далее.

## 6.2. Хранимые процедуры

Хранимая процедура является набором инструкций, хранящимся на стороне СУБД. Набор инструкций обычно пишется в виде последовательности SQL-команд. ХП похожи на обыкновенные процедуры языков высокого уровня, у них могут быть входные и выходные параметры и использоваться локальные переменные. В них могут производиться числовые вычисления и операции над символьными данными, результаты которых могут присваиваться переменным и параметрам. Кроме того, в ХП могут выполняться стандартные операции с БД. Особенно важно то, что в ХП возможны циклы и ветвления.

Использование ХП имеет следующие преимущества [11, 18]:

- *производительность*. При создании текст процедуры оптимизируется и хранится в БД в откомпилированном виде. В таком виде процедура выполняется гораздо быстрее, чем в случае динамического компилирования каждого составляющего ее запроса. Выполняются ХП сервером, а не клиентом, что позволяет сократить сетевой трафик;

- *модульное проектирование*. Использование хранимых процедур часто позволяет значительно сократить объём кода клиентского приложения. Приложения, получающие доступ к одной и той же БД, могут совместно использовать одни и те же процедуры, и не нужно снова программировать одни и те же действия, благодаря чему уменьшается риск программных ошибок в клиентских приложениях и экономится время программиста;

- *локализация изменений*. Если процедура модифицируется, то все внесенные изменения автоматически отражаются во всех приложениях, использующих процедуру, обеспечивая их согласованность;

- *защита*. В большинстве СУБД ХП считаются защищаемыми объектами и им назначаются отдельные привилегии. Пользователь, вызывающий ХП, должен иметь право на ее выполнение;

- *простота доступа*. В больших БД набор ХП может стать для прикладных программ основным средством доступа к БД. Часто вызов стандартной процедуры более понятен, чем выполнение последовательности SQL-команд (запросов);

- *реализация деловой логики*. Возможности условной обработки, предоставляемые ХП, часто используются для реализации бизнес-логики в БД. В хранимых процедурах можно реализовывать сложные алгоритмы обработки данных внутри базы, причем во многих случаях использование хранимых процедур позволяет чётко отделять алгоритмы логики программы от алгоритмов обработки данных.

### 6.2.1. Определение хранимых процедур

Хранимой процедурой (stored procedure) называется скомпилированная программа произвольной длины на процедурном языке СУБД, которая

хранится в БД как часть метаданных. Хранимые процедуры могут вызываться независимо - как в IBExpert, так и при выполнении SQL-сценария (в том числе в теле триггера и другой хранимой процедуры);

Различают создание ХП и их вызов.

В Firebird хранимые процедуры создаются запросом CREATE PROCEDURE, имеющим следующий синтаксис:

```
CREATE PROCEDURE имя_процедуры
  [(входной_параметр1 <тип_данных>
  [, входной_параметр2 <тип_данных> ... [= <значение>]]])
  [RETURNS (выходной_параметр1 <тип_данных> [, выходной_параметр1
<тип_данных> ...])]
  AS <тело_процедуры> [разделитель] .
```

Хранимая процедура может быть создана с помощью запроса CREATE PROCEDURE в SQL-редакторе IBExpert или в тексте SQL сценария.

Входные параметры хранимой процедуры указываются в скобках после имени создаваемой процедуры. Причем в качестве входных параметров могут использоваться выражения.

При объявлении входных параметров допускается задавать значения по умолчанию, например следующим образом:

```
CREATE PROCEDURE p1(x INTEGER = 123)
  RETURNS (y INTEGER) AS ...
```

**Примечание.** Параметры со значениями по умолчанию должны быть указаны последними в списке входных параметров, то есть нельзя объявить параметр, не имеющий значение по умолчанию, после параметров, объявленных со значением по умолчанию.

Подстановка значения по умолчанию происходит во время выполнения процедуры.

Предложение RETURNS предназначено для определения выходных параметров. Входные и выходные параметры могут использоваться в теле процедуры как обычные локальные переменные, определенные с помощью оператора DECLARE VARIABLE. Как уже было отмечено, в SQL-запросах локальные переменные, входные и выходные параметры должны использоваться с двоеточием перед именем переменной, чтобы отличать их от имен столбцов таблиц.

Тело процедуры и его составные части имеют следующие синтаксические конструкции:

```
<тело_процедуры> ::=
  [<список_объявления_переменных>]
  <блок_операторов>,
где
  <список_объявления_переменных> ::=
```

```
DECLARE [VARIABLE] имя_локальной_переменной <тип_данных> [{ = | DEFAULT}
<значение>];
[DECLARE [VARIABLE] имя_локальной_переменной <тип_данных> [{ = | DEFAULT}
<значение>]; ...] .
```

Для изменения определения процедуры используется запрос, имеющий следующий формат:

```
{ ALTER | RECREATE | CREATE OR ALTER } PROCEDURE имя_процедуры
[(входной_параметр1 <тип_данных>
[,входной_параметр2 <тип_данных> ...[= <значение>] ])]
[RETURNS (выходной_параметр1 <тип_данных> [,выходной_параметр2 <тип_данных>
...])]
AS <тело_процедуры> [разделитель] .
```

Если используется запрос ALTER PROCEDURE, то имя\_процедуры должно быть именем существующей процедуры. Это «мягкий» способ изменения кода процедуры, потому что если у нее есть зависимости (т.е. процедура используется другими объектами БД, например, другой ХП), на которые логически не влияют изменения, то они будут сохранены.

Запрос RECREATE PROCEDURE идентичен запросу CREATE PROCEDURE, за исключением того, что для существующей процедуры с тем же именем он сначала выполняет ее удаление перед созданием нового объекта. Следует учесть, что запрос RECREATE PROCEDURE не будет выполнен, если процедура используется другими объектами БД.

Запрос CREATE OR ALTER PROCEDURE создает процедуру, если она не существует, иначе изменяет определение существующей процедуры и перекомпилирует ее. При этом имеющиеся зависимости и привилегии сохраняются.

Удаление хранимых процедур осуществляется запросом DROP PROCEDURE, который имеет следующий формат:

```
DROP PROCEDURE имя_процедуры; .
```

Удалить хранимую процедуру можно лишь в том случае, если ее не используют другие процедуры или триггеры.

Хранимые процедуры Firebird условно подразделяются на два типа: *процедуры выбора* (select procedure) и *выполняемые процедуры* (executable procedure). Каждая из этих процедур имеет разное назначение. Создание ХП обоих типов формально не отличается, но отличается их вызов. Также отличаются операторы, которые влияют на ход выполнения ХП.

В хранимых процедурах выбора применяется оператор SUSPEND. Он *приостанавливает* выполнение процедуры для возвращения из нее текущих значений выходных параметров, после чего выполнение процедуры продолжается. Если выходным параметрам не присвоены значения, то будут

возвращены NULL значения. SUSPEND не должен использоваться в выполняемых процедурах, так как команды, следующие за ним, никогда не будут выполнены.

В выполняемых процедурах для выхода из цикла может использоваться оператор EXIT. Оператор EXIT вызывает переход на конечный END в процедуре и используется для того, чтобы прервать выполнение ХП и вернуться в точку ее вызова.

Как в ХП выбора, так и в выполняемых процедурах может использоваться оператор LEAVE, который служит для выхода из текущего цикла, а также для перехода на метку. Он позволяет приостановить выполнение текущего блока и перейти в место, помеченное соответствующей меткой. В этом случае используется следующий синтаксис:

```
....  
<метка>: <оператор_цикла>
```

```
...  
LEAVE [<метка>]
```

```
... ,
```

где <оператор\_цикла> представляет собой оператор WHILE, неявный курсор или оператор EXECUTE STATEMENT с циклом FOR;

<метка> – произвольный идентификатор, позволяющий именовать некоторый оператор модуля и таким образом сослаться на него. Допускается в качестве меток использование целых чисел без знака. Метка располагается непосредственно перед помечаемым оператором цикла и отделяется от него двоеточием. Переход на метку с помощью LEAVE <метка> осуществляется обычно по какому-то определенному условию, т.е. оператор LEAVE <метка> используется после фразы THEN или ELSE оператора ветвления IF. Причем оператор перехода LEAVE <метка> может быть вызван из цикла, вложенного в тот цикл, на который осуществляется переход.

Оператор LEAVE без явного указания метки означает выход из текущего цикла, например в теле ХП следующим образом:

```
...  
FOR SELECT RequestCD, COALESCE(ExecutionDate, 'Дата  
неизвестна')  
FROM Request  
INTO :Code, :Exec_Date  
DO  
BEGIN  
IF (Exec_Date = 'Дата неизвестна') THEN  
LEAVE; --выход из цикла выборки курсора  
ELSE  
SUSPEND;  
END
```

```
...
```

Для получения количества строк, возвращаемых выполненным запросом DML или запросом SELECT, предназначена контекстная переменная

ROW\_COUNT типа INTEGER. Она должна использоваться в том же блоке, что и запрос, например внутри тела ХП следующим образом:

```
...  
UPDATE Abonent SET Fio = :Fio WHERE AccountCD = :Account;  
IF (ROW_COUNT = 0) THEN  
    INSERT INTO Abonent (AccountCD, Fio) VALUES (:Account, :Fio);  
...
```

В данном примере выполняется запрос на обновление в таблице Abonent поля Fio новым значением для абонента с заданным номером лицевого счета (номер лицевого счета :Account и ФИО :Fio – входные параметры). Если строки с заданным номером лицевого счета нет в таблице, то контекстная переменная ROW\_COUNT принимает значение ноль и далее выполняется вставка строки в таблицу с данными номером лицевого счета и фамилией.

Контекстная переменная ROW\_COUNT также может применяться в цикле выборки (FETCH) при использовании явного курсора, как уже было показано ранее при изучении курсоров.

### 6.2.2. Процедуры выбора

Процедуры выбора возвращают результирующий набор строк, включающих столбцы, выбранные из одной или нескольких таблиц или представлений.

Использование процедур выбора позволяет объединять данные, получаемые несколькими запросами, и представлять полученные данные в виде единого набора выходных параметров процедуры выбора.

Например, пусть стоит задача получения какой-то информации из таблиц БД и для этой цели используется запрос с группировкой по какому-то полю внешнего ключа, представляющего собой код, значение которого расшифровывается в другой таблице. В запросе с группировкой не удастся включить в список возвращаемых элементов расшифровку значения кода внешнего ключа из другой таблицы (например, название улицы по внешнему ключу в таблице абонентов учебной БД). Если использовать в хранимой процедуре два последовательно выполняемых запроса – один на группировку, другой на расшифровку кода, а затем возвращать полученные результаты в виде одного набора данных, то, во-первых, можно упростить получение какой-либо аналитической информации (в данном примере), и, во-вторых, оптимизировать этот процесс за счет использования хранимой процедуры.

Для выполнения процедуры выбора используется запрос SELECT, в предложении FROM которого указывается имя ХП. Синтаксис запроса SELECT в этом случае имеет следующий вид:

```
SELECT <список_возвращаемых_элементов>  
FROM имя_процедуры ([ входной_параметр1 [,входной_параметр2 ...]])  
[WHERE <условие_поиска>]  
[ORDER BY <элемент_сортировки1> [, <элемент_сортировки2>]...];
```



Рассмотрим следующий SQL сценарий, который создает хранимую процедуру выбора с именем ListAbonent в учебной базе данных:

```
CONNECT 'c:\sqlab.fdb' USER 'SYSDBA' PASSWORD 'masterkey';
SET TERM !! ;
CREATE PROCEDURE ListAbonent
  RETURNS (LAccountCD VARCHAR(6), LFio VARCHAR(20),
          LPayDate DATE, LPaySum NUMERIC(15,2), LPayMonth SMALLINT,
          LPayYear SMALLINT)
  AS BEGIN
  FOR SELECT  A.AccountCD,  A.Fio,  P.PayDate,  P.PaySum,
P.PayMonth, P.PayYear
  FROM Abonent A, PaySumma P
  WHERE A.AccountCD = P.AccountCD AND P.PaySum > 70
  INTO :LAccountCD, :LFio, :LPayDate, :LPaySum, :LPayMonth,
:LPayYear
  DO
  SUSPEND;
END !!
SET TERM ; !!
```

Процедура выбора ListAbonent в выходных параметрах возвращает строку, состоящую из номера лицевого счёта абонента (LAccountCD), фамилии абонента (LFio), даты оплаты (LPayDate), суммы оплаты (LPaySum), месяца (LPayMonth) и года (LPayYear), за которые производится оплата, с суммой, большей 70.

Оператор SUSPEND приостанавливает выполнение процедуры ListAbonent для возвращения из нее текущих значений выходных переменных, после чего выполнение процедуры продолжается.

Таким образом, в процедуре ListAbonent производятся следующие действия:

- запрос SELECT формирует соединение таблиц Abonent, PaySumma и возвращает таблицу результатов запроса;
- курсор FOR SELECT ... DO для каждой строки таблицы результатов, сформированной запросом SELECT, выполняет блок операторов, которые следуют за предложением DO. В данном случае выполняется один оператор SUSPEND. Он приостанавливает выполнение процедуры, чтобы передать через выходные параметры процедуры очередную строку из таблицы результатов запроса.

Чтобы вывести все значения, возвращаемые процедурой ListAbonent, необходимо выполнить следующий запрос:

```
SELECT * FROM ListAbonent;
```

Результат выполнения запроса к хранимой процедуре выбора ListAbonent представлен на рис. 6.2.

LACCOUNTCD	LFIO	LPAYDATE	LPAYSUM	LPAYMONTH	LPAYYEAR
115705	МИЩЕНКО Е.В.	03.10.2001	250,00	9	2001
115705	МИЩЕНКО Е.В.	06.10.2000	250,00	9	2000
443069	СТАРОДУБЦЕВ Е.В.	03.10.2001	80,00	9	2001
080047	ШУБИНА Т.П.	26.11.1998	80,00	10	1998
080047	ШУБИНА Т.П.	21.11.2001	80,00	10	2001

**Рис. 6.2.** Результат выполнения запроса к процедуре ListAbonent

Результат, возвращаемый данной процедурой, может быть легко получен с помощью запроса SELECT без оформления ХП. Однако использование даже такой несложной процедуры вместо обычного запроса SELECT дает ряд преимуществ (простота доступа, сокращение объёма кода клиентского приложения и другие, описанные ранее), не говоря уже о процедурах, реализующих более сложные алгоритмы.

Рассмотрим пример SQL-сценария изменения определения процедуры ListAbonent для добавления входных параметров Mes и God. В теле следующей процедуры входные параметры Mes и God используются для отбора строк, содержащих информацию об оплатах с суммой больше 70, произведённых абонентом за услуги газоснабжения за заданный месяц (Mes) указанного года (God):

```
CONNECT 'c:\sqlab.fdb' USER 'SYSDBA' PASSWORD 'masterkey';
SET TERM !! ;
ALTER PROCEDURE ListAbonent (Mes SMALLINT, God SMALLINT)
  RETURNS (LAccountCD VARCHAR(6), LFio VARCHAR(20),
           LPayDate DATE, LPaySum NUMERIC(15,2),
           LPayMonth SMALLINT, LPayYear SMALLINT)
AS BEGIN
  FOR SELECT A. AccountCD, A.Fio, P.PayDate, P.PaySum, P.PayMonth,
           P.PayYear
  FROM Abonent A, PaySumma P
  WHERE A. AccountCD = P. AccountCD
  AND P.PaySum > 70 AND :Mes = P.PayMonth AND :God = P.PayYear
  INTO :LAccountCD, :LFio, :LPayDate, :LPaySum, :LPayMonth, :LPayYear
  DO
  SUSPEND;
  END !!
SET TERM ; !!
```

Если процедура выбора определена с указанием входных параметров, то их задание при вызове процедуры в запросе SELECT является обязательным. Например, чтобы вывести все сведения об оплаченных суммах, больших 70, за сентябрь 2001 года, необходимо выполнить следующий запрос:

```
SELECT * FROM ListAbonent (9, 2001);
```

Результат выполнения запроса представлен на рис. 6.3.

LACCOUNTCD	LFIO	LPAYDATE	LPAYSUM	LPAYMONTH	LPAYYEAR
115705	МИЩЕНКО Е.В.	03.10.2001	250,00	9	2001
443069	СТАРОДУБЦЕВ Е.В.	03.10.2001	80,00	9	2001

**Рис. 6.3.** Результат выполнения запроса к модифицированной процедуре ListAbonent

Рассмотрим пример процедуры выбора, в которой используется обновляемый явный курсор для позиционированного удаления и обновления строк таблицы. Создадим процедуру Exec\_Req, которая путем позиционированной модификации строк таблицы Request удаляет все невыполненные ремонтные заявки, а все непогашенные ремонтные заявки преобразует в погашенные. При этом в таблицу Executor заносится дополнительная информация по исполнителям, для которых произведено удаление или погашение заявок. Соответствующий скрипт выглядит следующим образом:

```
CONNECT 'c:\sqllab.fdb' USER 'SYSDBA' PASSWORD 'masterkey';
ALTER TABLE Executor Add Info VARCHAR(40);
SET TERM !! ;
CREATE PROCEDURE Exec_Req
RETURNS (ECode INTEGER, EName VARCHAR(20), Info VARCHAR(40))
AS
DECLARE EDate DATE;
DECLARE Exec SMALLINT;
DECLARE Req CURSOR FOR
(SELECT ExecutorCD, ExecutionDate, Executed FROM Request);
BEGIN
OPEN Req;
WHILE (1=1) DO
BEGIN
FETCH Req INTO :ECode, :EDate, :Exec;
IF (ROW_COUNT=0) THEN LEAVE;
IF (EDate IS NULL) THEN
BEGIN
DELETE FROM Request WHERE CURRENT OF Req;
UPDATE Executor SET Info = 'DEL_NotExec'
WHERE ExecutorCD=:ECode;
END
ELSE
IF (Exec=0) THEN
BEGIN
UPDATE Request SET Executed = 1 WHERE CURRENT OF Req;
UPDATE Executor SET Info = 'UPD_Executed'
```

```

        WHERE ExecutorCD=:ECode;
    END
END
CLOSE Req;
FOR
    SELECT ExecutorCD, Fio, Info
    FROM Executor
    INTO :ECode, :ENAME, :Info
    DO SUSPEND;
END !!
SET TERM ; !!

```

В начале скрипта с помощью запроса ALTER TABLE в таблицу Executor добавляется столбец Info. После этого создается процедура Exec\_Req, в которой сначала явным курсором Req выполняется выборка строк из таблицы Request, а затем проверяется ряд условий.

Если очередная ремонтная заявка не выполнена (EDate IS NULL), то происходит ее позиционированное удаление, а в таблице Executor поле Info получает значение 'DEL\_NotExec' (признак того, что для данного исполнителя удалены невыполненные заявки).

Если выполненная ремонтная заявка не погашена (Exec=0), то происходит позиционированное обновление (Executed = 1) той строки таблицы Request, на которой в данный момент находится курсор Req. Затем в таблице Executor происходит обновление поля Info. Поле Info получает значение 'UPD\_Executed' (признак того, что для данного исполнителя погашены непогашенные заявки).

Если очередная ремонтная заявка и выполнена, и погашена, то позиционированная модификация строки в таблице Request не выполняется и для исполнителя соответствующей заявки в таблице Executor поле Info не обновляется (будет иметь NULL-значение).

Затем курсор Req закрывается и происходит выборка в цикле FOR SELECT ... DO (неявный курсор) всех полей таблицы Executor в выходные параметры процедуры Exec\_Req.

Процедура Exec\_Req выполняется с помощью следующего запроса:

```
SELECT * FROM Exec_Req;
```

Результат представлен на рис. 6.4.

<b>ECODE</b>	<b>ENAME</b>	<b>INFO</b>
1	СТАРОДУБЦЕВ Е.М.	UPD_Executed
2	БУЛГАКОВ Т.И.	DEL_NotExec
3	ШУБИН В.Г.	UPD_Executed
4	ШЛЮКОВ М.К.	DEL_NotExec
5	ШКОЛЬНИКОВ С.М.	<null>

**Рис. 6.4.** Результат работы процедуры Exec\_Req

В таблице Request будут удалены две строки, где значение поля ExecutionDate равно NULL (ремонтные заявки с кодами 5 и 16), а в строках с кодами заявок 3 и 10 значение поля Executed примет значение 1.

### 6.2.3. Выполняемые процедуры

Выполняемые процедуры производят некоторые действия и могут не возвращать результирующий набор строк, как это происходит в процедурах выбора. Синтаксис определения выполняемых процедур аналогичен синтаксису определения процедур выбора с тем исключением, что выполняемые процедуры могут не содержать выходных параметров, а для правильной работы процедуры выбора наличие выходных параметров обязательно.

Запрос на вызов выполняемых хранимых процедур имеет следующий формат:

```
EXECUTE PROCEDURE имя_процедуры [( [входной_параметр1]
                                [, входной_параметр2 ...] )]
[RETURNING_VALUES выходной_параметр1 [,выходной_параметр2 ...]];
```

В качестве входных параметров, как и при вызове процедуры выбора, могут использоваться выражения.

Предложение RETURNING\_VALUES предназначено для присвоения каким-либо переменным значений, возвращаемых вызываемой процедурой.

Рассмотрим следующий пример SQL-сценария, в котором создается таблица Ftable и две выполняемые процедуры с именами Factorial [26] и FactorialSet:

```
CONNECT 'c:\sql\lab.fdb' USER 'SYSDBA' PASSWORD 'masterkey';
CREATE TABLE Ftable
(Fnum INTEGER, Fvalue DOUBLE PRECISION);
SET TERM !! ;
CREATE PROCEDURE Factorial (Num INTEGER)
RETURNS (N_Factorial DOUBLE PRECISION)
AS
  DECLARE VARIABLE Num_Less_One INTEGER;
BEGIN
  IF (Num < 0 OR Num > 170) THEN EXIT;
  IF (Num = 1 OR Num = 0) THEN
  BEGIN
    N_Factorial = 1;
    EXIT;
  END
  ELSE
  BEGIN
    Num_Less_One = Num - 1;
```

```

EXECUTE PROCEDURE Factorial (Num_Less_One)
  RETURNING_VALUES N_Factorial;
N_Factorial = N_Factorial * Num;
EXIT;
END
END !!
CREATE PROCEDURE FactorialSet
  (Minf INTEGER, Maxf INTEGER)
AS
  DECLARE VARIABLE i INTEGER;
  DECLARE VARIABLE f DOUBLE PRECISION;
BEGIN
  DELETE FROM Ftable;
  IF (Maxf < Minf) THEN EXIT;
  i = Minf;
  WHILE ( i <= Maxf) DO
  BEGIN
    EXECUTE PROCEDURE Factorial (i)
      RETURNING_VALUES f;
    INSERT INTO Ftable (Fnum, Fvalue) VALUES (:i, :f);
    i = i + 1;
  END
END !!
SET TERM ; !!

```

Таблица Ftable предназначена для хранения значений факториала. В столбце Fnum хранится число, для которого вычисляется факториал (ограничение <170 и >0), а в столбце Fvalue – само значение факториала.

Для типа DOUBLE PRECISION (тип выходного параметра) возможно вычисление факториала (без переполнения разрядной сетки) для значения 170, поэтому введена дополнительная проверка входного значения.

Выполняемая процедура Factorial вычисляет значение факториала рекурсивным методом. Если на входе процедуры ноль или единица, то значение факториала (N\_Factorial) принимается равным единице и оператор EXIT осуществляет возврат выходного параметра и выход из процедуры. Если на входе целое положительное число, меньшее 170 и отличное от нуля или единицы, то происходит рекурсивное вычисление факториала с использованием следующего соотношения:

$$\text{Num!} = \text{Num} * (\text{Num}-1)!$$

Рекурсивный вызов процедуры осуществляется при вызове процедурой самой себя запросом EXECUTE PROCEDURE.

**Примечание.** При рекурсивном вызове допускается глубина вложенности не более 1000 [27].

Оператор вызова процедуры EXECUTE PROCEDURE может также использоваться и в SQL-редакторе IBExpert. Например, чтобы вызвать

процедуру Factorial в интерактивном режиме и передать ей в качестве параметра значение 170, необходимо выполнить следующий запрос:

```
EXECUTE PROCEDURE Factorial (170);
```

Результат вызова процедуры Factorial представлен на рис. 6.5.

```
----- Procedure executing results: -----  
  
N_FACTORIAL = 7,25741561530799E306
```

**Рис. 6.5.** Результат вызова процедуры Factorial

Удалить процедуру Factorial нельзя из-за того, что ее использует процедура FactorialSet.

Выполняемая процедура FactorialSet предназначена для вычисления значений факториала чисел из некоторого диапазона и занесения полученных результатов в таблицу Ftable. Процедура получает в качестве параметров первое (Minf) и последнее (Maxf) значения диапазона вычисления факториала. В теле процедуры осуществляется цикл от Minf до Maxf с шагом, равным единице. Для каждого значения параметра цикла (локальная переменная i) вычисляется значение факториала путем вызова процедуры Factorial. Число, для которого вычислен факториал, и значение факториала заносятся в таблицу Ftable. Например, чтобы вычислить пять значений факториала от 1 до 5, необходимо в интерактивном режиме выполнить следующий запрос:

```
EXECUTE PROCEDURE FactorialSet (1, 5);
```

Для просмотра результатов выполнения процедуры необходимо выполнить следующий запрос для таблицы Ftable:

```
SELECT * FROM Ftable;
```

Результат выполнения запроса представлен на рис. 6.6.

Fnum	Fvalue
1	1,000
2	2,000
3	6,000
4	24,000
5	120,000

**Рис. 6.6.** Таблица Ftable со значениями факториалов от 1 до 5

В хранимых процедурах можно использовать запросы модификации данных INSERT, UPDATE OR INSERT, UPDATE и DELETE, содержащие предложение RETURNING для указания, что значения определенных столбцов в изменяемых строках должны запоминаться в соответствующих переменных, перечисленных в списке после INTO. Предложение RETURNING, употребляемое в конце запросов модификации DML, имеет следующий формат:

...  
RETURNING <список\_столбцов> INTO <список\_переменных>;

Столбцы, указанные в <список\_столбцов>, могут быть любыми столбцами из целевой таблицы, в том числе обновляемыми.

Следует учесть, что запросы модификации данных, использующие предложение RETURNING, должны оперировать не более чем с одной строкой набора данных.

Например, определение выполняемой хранимой процедуры, использующей запрос на удаление данных с предложением RETURNING, может выглядеть следующим образом:

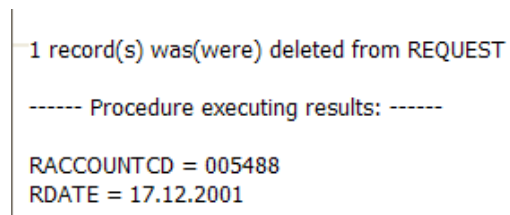
```
CREATE PROCEDURE Save_Del (Req_CD INTEGER)
RETURNS (RAccountCD VARCHAR(6), RDate DATE)
AS
BEGIN
DELETE FROM Request WHERE RequestCD= :Req_CD
RETURNING AccountCD, IncomingDate
INTO :RAccountCD, :RDate;
END.
```

Эта ХП удаляет информацию о ремонтной заявке с заданным номером (Req\_CD). При этом номер лицевого счета абонента, подавшего удаляемую заявку, и дата подачи этой заявки выводятся как результат выполнения процедуры.

Процедура может быть выполнена, например, с помощью следующего запроса:

```
EXECUTE PROCEDURE Save_Del(1);
```

Результат выполнения процедуры представлен на рис. 6.7.



```
1 record(s) was(were) deleted from REQUEST
----- Procedure executing results: -----
RACCOUNTCD = 005488
RDATE = 17.12.2001
```

**Рис. 6.7.** Результат выполнения процедуры Save\_Del

Процедурный язык Firebird предоставляет возможность выполнения запросов DDL и DML в теле модуля с помощью оператора EXECUTE STATEMENT. В таком случае запрос записывается как <строка>, которая может быть сконструирована как локальная переменная в теле модуля или же передана внешним приложением или хранимой процедурой в качестве входного аргумента для другой хранимой процедуры. Оператор EXECUTE STATEMENT может использоваться как в хранимых процедурах, так и в триггерах и выполняемых блоках на PSQL.

Синтаксис оператора EXECUTE STATEMENT следующий:



```
[FOR] EXECUTE STATEMENT <строка>
[INTO: <имя_переменной1> [, : <имя_переменной2> ...]] DO
<группа_операторов>;
```

Можно выделить в данном синтаксисе три формы использования оператора EXECUTE STATEMENT.

Первая форма (простейшая) представлена следующим синтаксисом:

```
EXECUTE STATEMENT <строка>;
```

В простейшей форме оператор EXECUTE STATEMENT выполняет запрос SQL, записанный как <строка>, который производит какое-то действие, но не возвращает строк данных. Такой запрос может представлять собой следующее:

- запрос DML (INSERT, DELETE, UPDATE);
- EXECUTE PROCEDURE;
- любой запрос DDL за исключением CREATE DATABASE и DROP DATABASE.

Например, если создать процедуру следующим образом:

```
CREATE PROCEDURE SampleOne
  (Sql VARCHAR(60), Account VARCHAR(30))
AS
BEGIN
  EXECUTE STATEMENT Sql || :Account;
END,
```

то выполнить ее можно, например, с помощью следующего запроса:

```
EXECUTE PROCEDURE SampleOne
  ('DELETE FROM REQUEST WHERE AccountCD=', '005488');
```

Вторая форма оператора EXECUTE STATEMENT имеет следующий формат:

```
EXECUTE STATEMENT <строка>
INTO :<имя_переменной1> [, :<имя_переменной2> ...].
```

В этом случае <строка> представляет собой запрос, возвращающий одиночную строку данных. Только <скалярный\_подзапрос> может быть выполнен при использовании второй формы оператора EXECUTE STATEMENT (<подзапрос\_столбца> и <табличный\_подзапрос> не могут использоваться).

Примером создания выполняемой процедуры, использующей вторую форму синтаксиса оператора EXECUTE STATEMENT, может быть следующий:

```
CREATE PROCEDURE SampleTwo (
  ColName VARCHAR(50),
  TableName VARCHAR (50))
RETURNS (Maxim NUMERIC(15,2))
```

```

AS
BEGIN
EXECUTE STATEMENT 'SELECT MAX(' || :ColName || ') FROM '
                || :TableName INTO :Maxim;
END.

```

Процедура `SampleTwo` определяет в указанной таблице БД максимальное значение в заданном столбце, имеющем числовой тип данных.

Процедура может быть выполнена следующим запросом:

```
EXECUTE PROCEDURE SampleTwo ('Paysum','Paysumma');
```

Результат выполнения процедуры `SampleTwo` представлен на рис. 6.8.

```

----- Procedure executing results: -----

MAXIM = 250

```

**Рис. 6.8.** Результат выполнения процедуры `SampleTwo`

Наконец, оператор `EXECUTE STATEMENT` поддерживает выполнение запроса `SELECT` внутри цикла `FOR` для возвращения в список переменных по одной строке за каждый проход цикла. Третья форма представлена следующим синтаксисом:

```

FOR EXECUTE STATEMENT <строка>
INTO :<имя_переменной1> [, :<имя_переменной2> ...]
DO
< группа_операторов >;

```

В данном случае `<строка>` может представлять собой запрос, возвращающий как одиночную строку данных, так и множество строк данных. Таким образом, любой запрос `SELECT` может быть использован в данной форме `EXECUTE STATEMENT`.

Рассмотрим процедуру, в которой используется оператор `EXECUTE STATEMENT` в третьей форме синтаксиса. Примером создания такой процедуры может быть следующий:

```

CREATE PROCEDURE SampleThree
(TextField VARCHAR(100), TableName VARCHAR(100))
RETURNS (Line VARCHAR(32000))
AS
DECLARE VARIABLE OneLine VARCHAR(100);
BEGIN
Line = "";
FOR EXECUTE STATEMENT 'SELECT ' || :TextField || ' FROM '
|| :TableName INTO :OneLine
DO
IF (OneLine IS NOT NULL) THEN

```

```
Line = Line || :OneLine || ', ' ;  
END.
```

Данная процедура производит запись в одну строку через запятую всех значений заданного столбца из указанной таблицы.

Процедура `SampleThree` может быть выполнена с помощью следующего запроса:

```
EXECUTE PROCEDURE SampleThree('Fio','Abonent');
```

Результат выполнения представлен на рис. 6.9.

```
----- Procedure executing results: -----
```

```
LINE = АКСЕНОВ С.А., МИЩЕНКО Е.В., КОНЮХОВ В.С., ТУЛУПОВА М.И., СВИРИНА З.А., СТАРОДУБЦЕВ Е.В., ШМАКОВ С.В., МАРКОВА В.П.,  
ДЕНИСОВА Е.К., ЛУКАШИНА Р.М., ШУБИНА Т.П., ТИМОШКИНА Н.Г.,
```

**Рис. 6.9.** Результат выполнения процедуры `SampleThree`

Оператор `EXECUTE STATEMENT` добавляет гибкости модулям на `PSQL`, но с риском ошибок. Оператор `EXECUTE STATEMENT` следует применять только в случае невозможности получить нужные результаты другими средствами или (что мало вероятно) когда это действительно улучшает выполнение запроса.

При использовании `EXECUTE STATEMENT` следует учесть следующие особенности:

- во время компиляции синтаксический анализатор не может проверить синтаксис запроса в строке аргумента;
- во время компиляции возвращаемые значения не проверяются на соответствие типов данных (например, строка '1234' может быть преобразована в целое число, а строка 'абв' вызовет ошибку преобразования), поэтому они должны быть внимательно проверены разработчиком, чтобы избежать непредсказуемых исключений преобразования данных при последующем выполнении процедуры;
- не проверяются зависимости или существование защиты для предотвращения удаления или изменения таблиц и столбцов;
- операции `EXECUTE STATEMENT` выполняются медленно, потому что встроенный оператор должен подготавливаться на сервере каждый раз перед выполнением;
- если ХП имеет специальные привилегии к некоторым объектам, то динамический оператор, выдаваемый в строке `EXECUTE STATEMENT`, не наследует их. Используются те привилегии, которые имеет пользователь, выполняющий процедуру.

Следует учесть также, что невозможно подсчитать количество строк, возвращаемых выполненным в операторе `EXECUTE STATEMENT` запросом с помощью переменной `ROW_COUNT`. Контекстная переменная `ROW_COUNT`, использованная после `EXECUTE STATEMENT`, всегда возвращает ноль.

## 6.3. Триггеры

Триггером называется особый вид хранимых процедур. В СУБД Firebird существует два вида триггеров: триггеры DML и триггеры БД.

**Триггеры DML** исполняются СУБД автоматически при проведении операций изменения данных. Триггер DML фактически является обработчиком на событие изменения данных в базе. При создании такой триггер связывается с какой-то одной таблицей БД. Триггеры DML предоставляют возможность производить некоторые определяемые пользователем действия при выполнении над данными заданной таблицей трех операций: удаления, изменения и вставки. Один и тот же триггер может отслеживать разные виды операций (например, вставку и изменение), причем операцию можно отслеживать до и после ее выполнения.

**Триггеры БД** не связываются с конкретной таблицей базы данных. Они исполняются при подключении к базе данных или отключении от нее, а также при запуске, успешном завершении или откате транзакции.

Использование триггеров имеет следующие преимущества:

- возможность автоматической проверки триггерами DML корректности данных, определенные значения которых необходимо хранить в соответствующей таблице;
- создание более гибкой системы поддержки целостности БД по сравнению со стандартными средствами;
- сокращение объема поддержки приложений, поскольку изменения триггеров автоматически отражаются во всех приложениях, использующих связанные с триггерами таблицы (без необходимости их повторной сборки и компиляции);
- возможность создания системы автоматической фиксации изменений в таблицах БД (ведение журнала изменений).

### 6.3.1. Триггеры DML

#### 6.3.1.1. Определение триггера

С любым событием, вызывающим изменение содержимого таблицы (представления), можно связать определенные действия, которые СУБД будет автоматически выполнять при каждом возникновении события.

Триггер DML – это предварительно определенное действие или последовательность действий, автоматически производимых при выполнении запросов обновления, добавления или удаления данных к определенной таблице (представлению) БД.

Триггер создается для заданной таблицы (представления) и сохраняется в БД как часть метаданных.

Триггер является мощным инструментом контроля изменений данных в БД, а также помогает автоматизировать операции, которые должны выполняться в этом случае. Триггер выполняется после проверки правил обновления данных.

Триггер включает в себя две компоненты:

- ограничения, для реализации которых он создается;
- событие, характеризующее возникновение ситуации, требующей проверки ограничений.

Предусмотренное действие производится за счет выполнения определенной операции или последовательности операций, с помощью которых реализуется логика, требуемая для удовлетворения ограничений.

События возникают при изменении содержимого таблицы. Действие, вызываемое событием, задается последовательностью запросов SQL и операторов процедурного языка Firebird.

Триггеры являются частью работы транзакции, в которой событие DML изменяет состояние строки. Если транзакция успешно подтверждается, то все действия триггеров принимаются. Если будет выполнен откат транзакции, то все действия триггера будут отменены.

В Firebird триггер определяется запросом CREATE TRIGGER, имеющим следующий формат:

```
CREATE TRIGGER имя_триггера
  FOR { базовая_таблица | представление }
  [ACTIVE | INACTIVE]
  { BEFORE | AFTER } <событие1> [OR <событие2> [OR <событие3>] ]
  [POSITION приоритет_триггера]
AS <тело_триггера> [ разделитель ],
где
<событие> ::= { DELETE | INSERT | UPDATE }.
```

Определение триггера состоит из заголовка и тела. Заголовок содержит:

- имя создаваемого триггера;
- имя таблицы или модифицируемого представления, для которых создается триггер;
- состояние триггера (активный или неактивный);
- описание связи с событиями, при наступлении которых триггер должен сработать;
- приоритет выполнения триггера над другими триггерами, если те связаны с той же таблицей (представлением).

В заголовке триггера его активность определяется указанием ключевого слова ACTIVE. Триггер можно "отключить", если использовать в заголовке триггера ключевое слово INACTIVE. По умолчанию любой триггер создается активным, т.е. не обязательно указывать ключевое слово ACTIVE при определении нового триггера. Явное указание ACTIVE может потребоваться при "включении" неактивного триггера (как можно модифицировать ранее созданный триггер будет подробно описано далее).

Триггер может выполняться в одной из двух фаз: BEFORE или AFTER. BEFORE указывает на то, что триггер должен сработать до указанных событий, а AFTER активизирует работу триггера после наступления указанных событий.

DELETE, INSERT и UPDATE задают три типа событий, на которые триггер "реагирует" соответственно при удалении, вставке или обновлении таблицы, для которой создан триггер.

Возможно объединение действий для двух или трех событий DML в одном триггере. В то же время в базе данных может быть создано несколько триггеров, которые ассоциированы с одной и той же таблицей, одним и тем же событием и имеют одну и ту же фазу. При этом для одной таблицы (представления) можно создать не более 32768 триггеров. Порядок выполнения таких триггеров устанавливается с помощью параметра предложения POSITION.

Значение приоритет\_триггера может меняться от 0 до 32767. Триггеры с меньшими номерами выполняются первыми. Если же несколько триггеров имеют один и тот же приоритет, то они выполняются в алфавитном порядке их имен.

Тело триггера задает последовательность действий, которая будет реализована СУБД при наступлении контролируемых событий над заданной таблицей или представлением.

Тело триггера, как и тело ХП, состоит из списка объявления используемых локальных переменных и блока выполняемых операторов:

```
<тело_триггера> ::= [<список_объявления_переменных>]  
<блок_операторов>,
```

где <список\_объявления\_переменных> и <блок\_операторов> имеют такой же синтаксис, который был описан ранее.

Рассмотрим пример триггера для одного события. Пусть требуется перед удалением записей в таблице Executor сохранять в таблице History информацию о ФИО всех исполнителей ремонтных заявок и дате, до которой данная информация была актуальна. Скрипт для создания триггера Executor\_Delete, реализующего решение данной задачи, будет выглядеть следующим образом:

```
CONNECT 'c:\sqlab.fdb' USER 'SYSDBA' PASSWORD 'masterkey';
```

```
/*создание в тексте SQL сценария таблицы History */
```

```
CREATE TABLE History (ID INTEGER PRIMARY KEY,  
    Executor_List VARCHAR(300), FixDate DATE);
```

```
/*создание генератора*/
```

```
CREATE SEQUENCE Gen_History;
```

```
/*установка начального значения генератора*/
```

```
ALTER SEQUENCE Gen_History RESTART WITH 0;
```

```
SET TERM !! ; /*установка нового разделителя*/
```

```
/* определение триггера */
```

```
CREATE TRIGGER Executor_Delete FOR Executor  
ACTIVE BEFORE DELETE
```

```
AS
```

```
DECLARE VARIABLE Line VARCHAR(32000);
```

```

BEGIN
  /* вызов хранимой процедуры для записи ФИО всех исполнителей в
  переменную Line */
  EXECUTE PROCEDURE SampleThree('Fio','Executor')
  RETURNING_VALUES Line;
/* вставка строки в таблицу History */
INSERT INTO History VALUES
  (NEXT VALUE FOR Gen_History, :Line, CURRENT_DATE);
END!! /*конец оператора CREATE TRIGGER */
SET TERM ; !! /*установка стандартного разделителя */.

```

В приведенном скрипте сначала производится создание таблицы History, а также определение и установка начального значения генератора Gen\_History, используемого для получения уникального значения поля первичного ключа таблицы History. Затем создается триггер Executor\_Delete, который будет запускаться перед наступлением события удаления строки из таблицы Executor. Триггер вызывает ранее созданную процедуру SampleThree, осуществляющую запись в переменную Line ФИО всех исполнителей ремонтных заявок, разделенных запятыми. Значение, полученное в переменной Line и дата, до которой информация была актуальна (дата удаления записей, т.е. текущая дата), записываются в таблицу History с помощью запроса INSERT, размещенного в теле триггера.

Следует обратить внимание на следующие особенности:

- отсутствие точки с запятой перед переменной Line в предложении RETURNING\_VALUES;
- использование двоеточия перед переменной Line в списке значений VALUES запроса INSERT. Двоеточие указывает на использование локальной переменной, а не имени столбца таблицы.

Характерной особенностью триггеров является то, что в них может использоваться ряд контекстных переменных, которые нельзя использовать в ХП.

Основным средством реализации возможностей триггеров по отслеживанию целостности данных являются контекстные переменные OLD и NEW. Переменные используются в следующем виде:

{OLD | NEW}.столбец .

Контекстная переменная OLD ссылается в строке на *текущие или предыдущие* значения, которые обновляются или удаляются соответственно запросами UPDATE или DELETE в таблице, для которой создан триггер. Таким образом, переменная OLD не используется при вставке, т.к. старого значения не существует.

Переменная OLD является контекстной переменной только для чтения, поэтому попытка присвоить какое-либо значение переменной OLD.столбец будет отклонена.

Контекстная переменная NEW ссылается в строке на *новые значения*, которые будут вставлены или обновлены соответственно запросами INSERT

или UPDATE для таблицы, для которой создан триггер. Переменная NEW не используется при удалении, т.к. новое значение не создается, а наоборот - происходит удаление.

Переменная NEW является контекстной переменной только для чтения в триггерах, которые срабатывают после наступления событий (фаза триггера – AFTER).

Контекстные переменные OLD и NEW могут использоваться в триггерах для нескольких событий. Но при этом могут возникнуть ситуации, когда для одного события, на которое реагирует триггер, контекстная переменная может использоваться, а для другого – нет. В этом случае ссылка на NEW.столбец в контексте удаления или на OLD.столбец в контексте добавления ошибкой не является – просто будет возвращено NULL-значение.

Рассмотрим пример создания триггера, контролирующего вставку строк в таблицу Request. Допустим, существует такое ограничение, что каждый исполнитель может быть назначен на выполнение не более 5 ремонтных заявок в год. При попытке зарегистрировать для исполнителя "лишнюю" ремонтную заявку должно выдаваться сообщение. Скрипт создания триггера для решения данной задачи будет выглядеть следующим образом:

```
/*подключение к БД */
```

```
CONNECT 'c:\sql\lab.fdb' USER 'SYSDBA' PASSWORD 'masterkey';
```

```
SET TERM !! ; -- определение нового разделителя
```

```
/* определение триггера*/
```

```
CREATE TRIGGER Num_Request FOR Request
```

```
ACTIVE BEFORE INSERT POSITION 0
```

```
AS
```

```
DECLARE ExecKod SMALLINT;
```

```
DECLARE God SMALLINT;
```

```
DECLARE NumRows SMALLINT;
```

```
BEGIN
```

```
FOR
```

```
SELECT ExecutorCD, EXTRACT(YEAR FROM IncomingDate), COUNT(*)
```

```
FROM Request
```

```
GROUP BY 1,2
```

```
INTO :ExecKod, :God, :NumRows
```

```
DO
```

```
BEGIN
```

```
IF (ExecKod = New.ExecutorCD AND
```

```
God = EXTRACT(YEAR FROM NEW.IncomingDate) AND
```

```
NumRows>=5)
```

```
/*генерация исключительной ситуации, если для исполнителя в указанном году  
уже зарегистрировано 5 заявок*/
```

```
THEN Exception Ins_Restrict;
```

```
END
```

```
END !! --конец оператора CREATE TRIGGER
```

```
SET TERM ; !! /*установка стандартного разделителя */
```



COMMIT; -- фиксация текущей транзакции.

Созданный триггер Num\_Request при попытке назначить исполнителю на выполнение более 5 заявок в год вызывает исключение Ins\_Restrict, определенное при создании учебной базы данных, и предотвращает вставку строки в таблицу Request.

В триггерах для нескольких событий с целью поддержания условных переходов между событиями используются такие логические контекстные переменные, как UPDATING, INSERTING, DELETING типа BOOLEAN. Они используются для задания определенных действий в зависимости от типа события DML, доступны в качестве предикатов в условных структурах, выполняющих операции изменения состояния данных, например, в теле триггера в следующем виде:

```
IF (INSERTING OR DELETING) THEN  
    NEW.ID = GEN_ID(G_GENERATOR_1, 1);
```

Рассмотрим пример триггера для нескольких событий. Пусть требуется в некоторой таблице Journal фиксировать имена пользователей, которые добавляют, обновляют или удаляют данные в таблице NachislSumma, идентификатор факта начисления(NachislFactCD), признак типа события ('INS' для вставки, 'EDIT' для обновления и 'DEL' для удаления), а также дату и время совершения операции. Скрипт для создания такого триггера с именем Usr\_Action будет выглядеть следующим образом:

```
/*подключение к БД */  
CONNECT 'c:\sqlab.fdb' USER 'SYSDBA' PASSWORD 'masterkey';  
/*создание таблицы Journal */  
CREATE TABLE Journal (Usr_Name VARCHAR(20), Nach_ID INTEGER,  
DML_Event VARCHAR(4), Time_Date TIMESTAMP);  
SET TERM !! ; /*установка нового разделителя*/  
/* определение триггера */  
CREATE TRIGGER Usr_Action FOR NachislSumma  
ACTIVE AFTER INSERT OR UPDATE OR DELETE  
AS  
DECLARE VARIABLE Nach INTEGER;  
DECLARE VARIABLE Event VARCHAR(4);  
BEGIN  
    IF (DELETING) THEN  
        BEGIN  
            Nach = OLD.NachislFactCD;  
            Event = 'DEL';  
        END  
    ELSE  
        BEGIN  
            Nach = NEW.NachislFactCD;  
            IF (UPDATING) THEN  
                Event = 'EDIT';
```

```

ELSE
  Event = 'INS';
END
/* вставка строки в таблицу Journal */
INSERT INTO Journal VALUES
(USER, :Nach, :Event, CURRENT_TIMESTAMP);
END!! /*конец оператора CREATE TRIGGER */
SET TERM ; !! /*установка стандартного разделителя */.

```

Чтобы проверить действие созданного триггера `Usr_Action`, от имени пользователя `SYSDBA` выполним последовательно три следующих запроса:

```
INSERT INTO NachisSumma VALUES (51,'005488',1,63,1,2002);
```

```
DELETE FROM NachisSumma WHERE NachisFactCD=1;
```

```
UPDATE NachisSumma SET GazServiceCD=2
WHERE NachisFactCD=40;
```

Данные, помещенные в таблицу `Journal` и полученные в результате выполнения следующего запроса:

```
SELECT * FROM Journal; ,
```

представлены на рис. 6.10.

USR_NAME	NACH_ID	DML_EVENT	TIME_DATE
SYSDBA	51	INS	04.06.2007 23:00
SYSDBA	1	DEL	04.06.2007 23:01
SYSDBA	40	EDIT	04.06.2007 23:03

**Рис. 6.10.** Данные таблицы `Journal`

Описание всех имеющихся в БД триггеров (как системных, так и пользовательских) хранится в системной таблице `RDB$TRIGGERS`. При использовании утилиты `IBExpert` все пользовательские триггеры отображаются в окне инспектора объектов БД.

### 6.3.1.2. Примеры поддержания ссылочной целостности

Рассмотрим примеры поддержания целостности БД с помощью триггеров.

Пусть необходимо написать SQL сценарий по созданию триггера, предотвращающего вставку `NULL` значений в поля внешних ключей `AccountCD`, `ExecutorCD` и `FailureCD` таблицы `Request`. Если значения, вводимые в поля перечисленных внешних ключей, не содержат `NULL`, то необходимо вставить в поле первичного ключа `RequestCD` очередное значение, полученное с помощью генератора. При попытке вставить `NULL` значение должен сработать триггер и выдать сообщение об исключительной ситуации, т.е.

триггер должен запускаться при каждой вставке строки в таблицу Request. Ниже приведен текст SQL сценария, реализующего поставленную задачу.

```
/*подключение к БД */
CONNECT 'c:\sql\lab.fdb' USER 'SYSDBA' PASSWORD 'masterkey';
/* создание сообщения об исключительной ситуации*/
CREATE EXCEPTION E_Request 'Error insert null value into Request ';
/*создание генератора*/
CREATE GENERATOR Request_GEN;
/*установка начального значения генератора*/
SET GENERATOR Request_GEN TO 24;
SET TERM !! ; -- определение нового разделителя
CREATE TRIGGER Request_INSERT FOR Request
BEFORE INSERT AS
/*конец заголовка и начало тела триггера*/
BEGIN
/*проверка значений вставляемой строки в таблицу Request; при этом
переменная NEW представляет собой ссылку на вставляемое значение в
указанное поле внешнего ключа таблицы Request */
    IF ((NEW.AccountCD IS NULL) OR (NEW.ExecutorCD IS NULL) OR
        (NEW.FailureCD IS NULL))
    THEN
        /*генерация исключительной ситуации, если значения полей AccountCD,
        ExecutorCD или FailureCD содержат NULL значения*/
        EXCEPTION E_Request;
    ELSE
        /*присвоение полю первичного ключа таблицы Request очередного значения
        генератора Request_GEN */
        NEW.RequestCD = GEN_ID(Request_GEN, 1);
    END !!      --конец оператора CREATE TRIGGER
SET TERM ; !! /*установка стандартного разделителя */
COMMIT;      -- фиксация текущей транзакции
```

Следует обратить внимание на способ присвоения полю RequestCD (первичный ключ таблицы Request) очередного значения: контекстная переменная NEW используется в качестве имени таблицы (Request), в которую вставляется строка. В данном случае нельзя вставлять внутри тела триггера значение первичного ключа в таблицу Request с помощью запроса INSERT (т.к. этот триггер создан именно для события INSERT таблицы Request, то он фактически будет пытаться вызвать сам себя). Таким образом, вставка очередной строки в таблицу Request должна производиться запросом INSERT без указания значения первичного ключа, которое будет генерироваться автоматически, например в следующем виде:

```
INSERT INTO Request (AccountCD, ExecutorCD, FailureCD,
IncomingDate, ExecutionDate, Executed)
```

```
VALUES ('005488', 3, 1, '28.12.2005', '01.02.2006', 0);
```

При выполнении данного запроса срабатывает триггер Request\_INSERT, т.к. он создан для таблицы Request, активный и имеет реакцию на тип события INSERT. Так как вставляемая строка не содержит NULL значения, то запрос будет выполнен успешно. Если же будет произведена попытка вставки NULL-значений в поля внешних ключей таблицы Request, то сгенерируется исключительная ситуация E\_Request, будет выдано сообщение Error insert null value into Request и вставки не произойдет.

Рассмотрим еще один пример. Пусть необходимо создать условие ссылочной целостности с помощью триггера, который бы запрещал удаление и изменение значения первичного ключа StreetCD в таблице Street при наличии ссылающихся на него внешних ключей в таблице Abonent. Ниже приведен текст SQL сценария, реализующего поставленную задачу.

```
CONNECT 'c:\sql\lab.fdb' USER 'SYSDBA' PASSWORD 'masterkey';
SET TERM !! ;
CREATE TRIGGER Str_DELETE FOR Street
BEFORE DELETE OR UPDATE
AS
DECLARE VARIABLE NumRows INTEGER;
BEGIN
    SELECT COUNT(*) FROM Abonent
    WHERE Abonent.StreetCD = OLD.StreetCD INTO :NumRows;
    IF (NumRows > 0) THEN
        BEGIN
            IF (DELETING)
                THEN EXCEPTION Del_Restrict;
            IF (UPDATING)
                THEN EXCEPTION Upd_Restrict;
        END
    END!!
SET TERM ; !!.
```

Созданный триггер с именем Str\_DELETE выполняется перед удалением или обновлением строки в таблице Street. С помощью агрегатной функции COUNT(\*) подсчитывается количество строк в таблице Abonent, которые имеют внешние ключи, ссылающиеся на удаляемый или обновляемый первичный ключ таблицы Street. Если количество строк не равно нулю, то анализируется вид события DML. Если это удаление записи, то возникает определенная при создании учебной базы данных исключительная ситуация Del\_Restrict, прерывающая выполнение тела триггера, и удалить первичный ключ из таблицы Street, который имеет ссылающийся на него внешний ключ, становится невозможно. Если это обновление записи, то возникает исключительная ситуация Upd\_Restrict, прерывающая выполнение тела триггера, и обновить первичный ключ из таблицы Street, который имеет

ссылающийся на него внешний ключ, становится невозможно. Таким образом, созданный триггер реализует правило ссылочной целостности, которое запрещает удаление и обновление первичного ключа родительской таблицы, если имеются ссылающиеся на него внешние ключи дочерней таблицы.

### 6.3.1.3. Модификация и удаление триггера

Триггер представляет собой весьма полезное средство, но в то же время триггеры необходимо очень тщательно отлаживать, так как неправильно написанные триггеры могут привести к серьезным проблемам. При неправильной логике работы триггеров можно легко уничтожить нужные данные и даже целую базу данных.

Часто возникает необходимость модифицировать ранее созданный триггер DML. Причинами этого могут быть:

- исправление ошибок, допущенных в процессе разработки триггера и нарушающих правильную логику работы системы;
- изменение функциональности уже созданного триггера;
- временное отключение триггера, полезное при разработке и отладке.

Производить модификацию триггера можно в тексте SQL сценария или путем выполнения отдельного запроса на модификацию в SQL-редакторе IBExpert.

Для изменения определения созданного триггера DML необходимо использовать запросы ALTER TRIGGER, CREATE OR ALTER TRIGGER или RECREATE TRIGGER, которые имеют следующий формат:

```
{ ALTER TRIGGER имя_триггера
| { CREATE OR ALTER | RECREATE } TRIGGER имя_триггера
  FOR { базовая_таблица | представление } }
[ACTIVE | INACTIVE]
[ { BEFORE | AFTER } <событие1> [OR <событие2> [OR <событие3> ] ] ]
[POSITION приоритет_триггера]
  [AS <тело_триггера>] [разделитель].
```

Изменения триггера могут быть следующих трех видов:

- только заголовка;
- только тела;
- заголовка и тела.

Если требуется изменить только заголовок триггера, то можно использовать запрос ALTER TRIGGER. Данный запрос требует хотя бы одного изменяемого атрибута после имени триггера. Любой атрибут заголовка, опущенный в этом запросе, остается неизменным. Если изменяется индикатор фазы, то событие также должно быть указано. Как следует из приведенного выше синтаксиса, запрос ALTER TRIGGER не может связать триггер с другой таблицей, отличной от заданной ранее при создании триггера.

Изменение заголовка триггера может быть использовано, например, для отключения триггера в следующем виде:

```
ALTER TRIGGER Str_DELETE INACTIVE;
```

Изменение тела триггера используется, как правило, для преобразования его функциональности. Например, созданный ранее триггер Request\_INSERT можно изменить, убрав генерацию значения первичного ключа следующим образом:

```
ALTER TRIGGER Request_INSERT AS
BEGIN
  IF ((NEW.AccountCD IS NULL) OR (NEW.ExecutorCD IS NULL)
      OR (NEW.FailureCD IS NULL))
  THEN
    EXCEPTION E_Request;
  END.
```

Изменение (точнее, замещение) всего триггера (как заголовка, так и тела) следует реализовывать в тексте SQL сценария. При этом можно использовать запрос ALTER TRIGGER (применение запроса CREATE TRIGGER вызовет ошибку, так как к моменту изменения триггер уже существует).

Часто удобнее использовать запрос CREATE OR ALTER TRIGGER, который создает триггер, если он еще не существует, или изменяет триггер с указанным именем и перекомпилирует его. При этом имеющиеся зависимости и привилегии сохраняются.

Если требуется заново создать триггер со старым именем, то используется запрос RECREATE TRIGGER. Синтаксис этого запроса такой же, как и запроса CREATE TRIGGER. Если триггер с указанным именем уже существует, то запрос RECREATE TRIGGER пытается удалить его и создать полностью новый объект (что не будет выполнено, если триггер находится в использовании).

Для удаления триггера используется запрос DROP TRIGGER, который имеет следующий формат:

```
DROP TRIGGER имя_триггера;
```

Например, чтобы удалить триггер Str\_DELETE и правило ссылочной целостности, которое связано с этим триггером, необходимо применить следующий запрос:

```
DROP TRIGGER Str_DELETE;
```

### 6.3.2. Триггеры базы данных

Как было отмечено ранее, кроме триггеров DML существуют триггеры БД [20]. Они предоставляют возможность производить определяемые пользователем действия при выполнении подключения к БД и отключении от нее, а также при запуске, успешном завершении или откате транзакции.

Синтаксис запроса на создание или изменение триггера БД имеет следующий вид:

```
{CREATE | RECREATE | CREATE OR ALTER} TRIGGER имя_триггера
[ACTIVE | INACTIVE]
ON <событие>
[POSITION приоритет_триггера]
AS <тело_триггера>,
где <событие> ::= {CONNECT | DISCONNECT | TRANSACTION START
                  | TRANSACTION COMMIT
                  | TRANSACTION ROLLBACK }.
```

События CONNECT и DISCONNECT возникают при выполнении команд подключения к БД и отсоединения от БД, которые были описаны ранее при изучении SQL-скриптов. Событие TRANSACTION START возникает при запуске новой транзакции, т.е. автоматически вместе с первым SQL-запросом или непосредственно после окончания предыдущей транзакции. События TRANSACTION COMMIT и TRANSACTION ROLLBACK возникают при выполнении команд COMMIT или ROLLBACK соответственно, которые означают завершение транзакции (фиксацию или откат) и будут подробно рассмотрены далее.

Если сравнить данный синтаксис с синтаксисом определения триггеров DML, то можно выделить следующие особенности:

- не указывается конкретная таблица (представление), для которой создается триггер;
- в качестве событий используются подключение к БД, отключение от БД, старт, успешное завершение и откат транзакций;
- отсутствует указание фазы BEFORE или AFTER для событий.

Следует отметить, что только владелец БД или SYSDBA могут создавать триггеры БД, в то время как триггер DML может быть определен любым другим пользователем, если он является владельцем таблицы, для которой создается триггер.

Рассмотрим пример создания триггера БД Test\_Connect, заносящего в таблицу Users\_Connect имена пользователей, подключающихся к учебной БД, а также дату и время подключения. Скрипт будет выглядеть следующим образом:

```
CREATE TABLE Users_Connect
  (User_Name VARCHAR(20), Date_Time TIMESTAMP);
SET TERM !! ;
CREATE TRIGGER Test_Connect
ACTIVE
ON CONNECT
AS
BEGIN
  INSERT INTO Users_Connect
```

```
VALUES (USER, CURRENT_TIMESTAMP);
END!!
SET TERM ; !!
COMMIT;
```

В результате выполнения данного скрипта при каждом подключении к учебной БД в таблицу Users\_Connect будут занесены имя подключающегося пользователя и дата/время на момент подключения.

Аналогично описанной ранее модификации триггеров DML может быть выполнено изменение активности или изменение тела ранее созданного триггера БД. Однако следует учесть, что событие, с которым ассоциирован триггер БД, не может быть изменено.

## 6.4. Выполняемые блоки

Как уже отмечалось, существует возможность выполнения кода на PSQL без оформления именованных ХП или триггеров. Такую возможность предоставляет запрос EXECUTE BLOCK, который имеет следующий синтаксис:

```
EXECUTE BLOCK [(входной_параметр1 <тип_данных> = : входной_параметр1
               [, входной_параметр2 <тип_данных> = : входной_параметр2])]
[RETURNS (выходной_параметр1 <тип_данных>
          [, выходной_параметр2 <тип_данных> ...])]
AS <тело_блока>.
```

Данный оператор может использоваться в клиентских приложениях для реализации требуемой логики работы системы, когда в используемой базе данных нет необходимой именованной ХП или триггера. Входные параметры в таком случае передаются в оператор EXECUTE BLOCK из приложения, а выходные – возвращаются клиентскому приложению. В IBExpert данный оператор можно выполнить в SQL-редакторе как обычный запрос.

В операторе EXECUTE BLOCK <тело\_блока> может включать в себя все те же конструкции (операторы, запросы), что и тело хранимой процедуры или триггера.

При изучении выполняемых процедур была рассмотрена процедура SampleThree, выполняющая запись в одну строку всех значений заданного столбца из указанной таблицы. Реализовать ту же самую операцию без создания именованной ХП можно следующим выполняемым блоком:

```
EXECUTE BLOCK (TextField VARCHAR(50)=:TextField,
              TableName VARCHAR(50)=:TableName)
RETURNS (Line VARCHAR(10000))
AS
DECLARE VARIABLE OneLine VARCHAR(50);
BEGIN
```



```

Line = "";
FOR EXECUTE STATEMENT 'SELECT ' || :TextField || ' FROM '
|| :TableName INTO :OneLine DO
IF (OneLine IS NOT NULL) THEN
    Line = Line || OneLine || ', ';
SUSPEND;
END.

```

Результат выполнения блока для столбца Fio таблицы Abonent совпадает с результатом, представленным на рис. 6.9.

## Контрольные вопросы

1. В чем состоят преимущества использования хранимых процедур в языке SQL?
2. В чем состоят преимущества использования триггеров?
3. Как осуществляется объявление локальной переменной в теле модуля на процедурном SQL?
4. Какие условные операторы используются в процедурном SQL?
5. Что такое курсор? Каковы особенности работы с явным и неявным курсором?
6. Как запомнить значения определенных столбцов изменяемых таблиц в переменных при использовании запросов модификации данных в PSQL?
7. Что такое генератор последовательности? Как его создать, использовать и удалить с помощью средств языка SQL?
8. Что такое исключение? Как оно создается, изменяется и вызывается?
9. С помощью какого оператора осуществляется динамическое выполнение запросов DDL и DML в модуле процедурного SQL? Какие используются формы этого оператора?
10. Что такое SQL-сценарий? Как выполнить создание БД в SQL-сценарии?
11. Что такое триггер DML? Как он создается, модифицируется и удаляется?
12. Как создать триггер БД? Чем триггер БД отличается от триггера DML?
13. В чем состоят отличия семантического характера хранимых процедур от триггеров? Как создать, изменить и удалить ХП?
14. Какие виды хранимых процедур существуют? Чем они отличаются друг от друга? Как они вызываются?
15. Как выполнить блок кода на процедурном языке без оформления хранимой процедуры или триггера?

## 7. Защита данных

В современных ИС важно не только правильно спроектировать структуру БД и манипулировать данными, но также и обеспечить защиту этих данных.

Защита данных – это мероприятия по охране данных от множества возможных угрожающих ситуаций, как преднамеренных, так и случайных.

Причинами возможного разрушения или потери данных могут быть:

- порча или изменение данных анонимным пользователем;
- завершение работы программ при системном сбое, когда база данных остается в непредсказуемом состоянии;
- возникновение конфликта при выполнении двух и более программ, конкурирующих за одни и те же данные;
- изменение базы данных недопустимым способом обновления и т.д.

Защита базы данных от подобных проблем реализуется, в основном, за счет управления доступом к данным, а также с помощью механизма транзакций.

Материал настоящей главы в большей мере ориентирован на администраторов базы данных. Здесь рассматривается система безопасности, принятая в языке SQL. Излагаются общие правила разграничения доступа пользователей к объектам базы данных и описываются методы управления доступом.

Также в данной главе приводится определение транзакции и ее свойств, рассматривается механизм сохранения и отката транзакций, описывается использование механизма транзакций для восстановления системы при повреждениях. Вводятся понятие параллельности в работе базы данных и методы управления параллельностью с использованием блокировок.

## **7.1. Управление доступом к данным**

### **7.1.1. Требования к безопасности данных**

В любой ИС существует конфиденциальная информация, доступ к которой может быть разрешен лишь ограниченному кругу лиц. Безопасность данных является важным аспектом управления базами данных, поскольку информация, хранимая в БД, представляет собой чрезвычайно ценный ресурс. Поэтому обеспечение безопасности хранимых данных является неотъемлемой частью любой современной СУБД, в том числе и Firebird.

Под *безопасностью* данных понимается защита данных от несанкционированного доступа.

Схема доступа к данным в реляционных СУБД базируется на трех следующих принципах.

1. **Пользователи СУБД** рассматриваются как основные действующие лица, желающие получить доступ к данным.

Безопасность целесообразно рассматривать в многопользовательской среде, когда существует определенный круг пользователей, имеющих доступ к БД.

Логически пользователь Firebird – это регистрационная (учетная) запись, доступная во всех базах данных, обслуживаемых сервером. Чтобы зарегистрировать нового пользователя, необходимо воспользоваться либо инструментом командной строки gsec, либо графическим инструментом администрирования БД. Например, чтобы зарегистрировать нового пользователя с помощью IVExpert, необходимо воспользоваться пунктом

«Менеджер пользователей» меню «Инструменты». С помощью SQL-команды создать или удалить пользователя Firebird нельзя – это следствие вынесения системы безопасности на уровень сервера. Имя пользователя может иметь длину до 31 символа включительно, пароль – до 32 символов, однако из них для аутентификации используются только первые 8 символов. Для имен пользователей не важен регистр символов, но пароль является регистрочувствительным.

Следует отметить, что только системный администратор SYSDBA может создавать новых пользователей в Firebird.

Как уже отмечалось, информация обо всех пользователях, которые имеют доступ к серверу Firebird, и их паролях хранится в файле *security2.fdb*. Когда удаленный или локальный клиент соединяется с БД, происходит идентификация пользователя.

Стабильная система управления пользователями – обязательное условие безопасности данных, хранящихся в любой реляционной СУБД.

**2. Объекты доступа** - это элементы базы данных, доступом к которым можно управлять (разрешать доступ или защищать от доступа). Обычно объектами доступа являются таблицы, однако ими могут быть и другие объекты БД – представления, хранимые процедуры и т.д.

Как правило, для обеспечения безопасности требуется, чтобы данные в любой таблице были доступны не всем пользователям, а лишь некоторым из них, а для определенных таблиц необходимо обеспечить выборочный доступ к ее столбцам. Некоторым пользователям должен быть запрещен непосредственный (через запросы) доступ к таблицам, но разрешен доступ к этим же таблицам с помощью механизма представлений.

Итак, конкретный пользователь обладает конкретными правами доступа к конкретному объекту.

**3. Привилегии** – это системный признак, определяющий операции, которые разрешено выполнять пользователю над конкретными объектами.

СУБД от имени конкретного пользователя выполняет операции над базой данных, то есть добавляет строки в таблицы (INSERT), удаляет строки (DELETE), обновляет данные в строках таблицы (UPDATE). Она делает это в зависимости от того, обладает ли конкретный пользователь правами на выполнение конкретных операций над конкретным объектом БД.

Например, некоторым пользователям может быть разрешено обновление данных в таблицах, в то время как для других допускается лишь выбор данных из этих же таблиц.

Таким образом, в СУБД авторизация доступа осуществляется с помощью привилегий. Установление и контроль привилегий являются обязанностью администратора базы данных.

### **7.1.2. Привилегии доступа и передача привилегий**

Создание учетной записи пользователя само по себе не дает ему никаких прав на доступ к объектам БД. Безопасность в языке SQL на уровне доступа к

таблице (представлению) управляется с помощью *привилегий доступа*, т.е. списком операций, которые пользователю разрешено выполнять над данной таблицей (представлением).

Для управления привилегиями служит запрос GRANT, который передает привилегии доступа определенному пользователю, роли, процедуре или триггеру к таблице или представлению. Для удаления привилегий используется запрос REVOKE.

Запрос GRANT может быть использован для следующих операций по передаче привилегий:

- передача пользователям, триггерам, хранимым процедурам или представлениям: SELECT, INSERT, UPDATE, DELETE и REFERENCES привилегий доступа к таблице;
- передача пользователям, триггерам, хранимым процедурам или представлениям: SELECT, INSERT, UPDATE и DELETE привилегий доступа к представлению;
- передача для роли: SELECT, INSERT, UPDATE, DELETE и REFERENCES привилегий доступа к таблице;
- передача для роли: SELECT, INSERT, UPDATE и DELETE привилегий доступа к представлению;
- передача пользователям права использовать привилегии, переданные конкретной роли, т.е. при подключении к БД обладать привилегиями, определенными соответствующей ролью в БД;
- передача пользователям, триггерам, хранимым процедурам или представлениям EXECUTE привилегии на выполнение хранимой процедуры.

В табл. 7.1 приведено описание привилегий доступа.

**Таблица 7.1.** Описание привилегий доступа

<b>Привилегия</b>	<b>Уровень доступа</b>
ALL	Выбор, вставка, обновление, удаление данных и ссылка внешнего ключа другой таблицы на первичный ключ данной таблицы
SELECT	Чтение данных
INSERT	Запись новых данных
UPDATE	Изменение существующих данных
DELETE	Удаление данных
EXECUTE	Выполнение или вызов хранимой процедуры
REFERENCES	Ссылка внешнего ключа другой таблицы на первичный ключ данной таблицы
Имя_роли	Представляет все привилегии, переданные данной роли

**Примечание.** Зарезервированное слово ALL обеспечивает механизм для присвоения SELECT, DELETE, INSERT, UPDATE и REFERENCES привилегий одновременно. При этом ALL не передает право на использование роли или EXECUTE привилегию на выполнение хранимой процедуры.

Все объекты БД имеют определенный уровень доступа при их создании. Первоначально только создатель объекта БД имеет к нему доступ и только он может передать определенные привилегии доступа другим пользователям или хранимым процедурам. При этом системный администратор Firebird SYSDBA имеет доступ *ко всем без исключения* объектам БД. В связи с этим *настоятельно* рекомендуется как можно раньше изменить стандартный пароль "masterkey", который имеет SYSDBA.

**Примечание.** СУБД не проверяет никакие права доступа для пользователя SYSDBA. В частности, поэтому операции над множеством объектов производятся от имени SYSDBA чуть быстрее, чем от имени других пользователей [18].

Запрос GRANT имеет следующий синтаксис:

```
GRANT {<привилегии> ON [TABLE] {базовая_таблица | представление}
      TO {<объект> | <список_пользователей> [WITH GRANT OPTION] } }
      | EXECUTE ON PROCEDURE имя_процедуры
      TO {<объект> | <список_пользователей> [WITH GRANT OPTION]}
      | <список_ролей>
      TO {PUBLIC | <список_пользователей> [WITH ADMIN OPTION]};
```

где

<привилегии> ::= {ALL [PRIVILEGES] | <список\_привилегий>;}

<список\_привилегий> ::=

```
{SELECT
 | DELETE
 | INSERT
 | UPDATE [( <список_столбцов> )]
 | REFERENCES [( <список_столбцов> )]}
[, <список_привилегий> ...];
```

<объект> ::=

```
{PROCEDURE имя_процедуры
 | TRIGGER имя_триггера
 | VIEW представление
 | [ROLE] имя_роли
 | PUBLIC [, <объект> ...]};
```

<список\_пользователей> ::=

```
[USER] имя_пользователя1 [, [USER] имя_пользователя2 ...];
```

<список\_ролей> ::= имя\_роли1 [, имя\_роли2 ...].

Зарезервированное слово PUBLIC означает, что указанные привилегии доступа к заданному объекту БД становятся *общедоступными* и невозможно отобрать эти привилегии у *конкретного* пользователя. Следует учесть, что PUBLIC используется для обозначения именно всех пользователей БД, и

переданные с помощью PUBLIC привилегии не будут доступны ХП, триггерам, представлениям и ролям.

Пусть на сервере зарегистрированы, например с помощью утилиты IBEExpert, два пользователя: Ivanov и Petrov. Имеются: представление Ab\_PaySum\_VIEW, хранимые процедуры Factorial и FactorialSet, триггеры Abonent\_DELETE и Services\_UPDATE, которые, как и все остальные объекты БД, созданы пользователем SYSDBA.

Например, для передачи пользователю Ivanov *всех привилегий* доступа на таблицу Abonent следует выполнить следующий запрос GRANT:

```
GRANT ALL ON Abonent TO Ivanov;
```

Передача привилегии SELECT на представление Ab\_PaySum\_VIEW пользователю Petrov может быть выполнена следующим образом:

```
GRANT SELECT ON Ab_PaySum_VIEW TO Petrov;
```

Для передачи в общее пользование права обновлять столбцы Fio и AccountCD таблицы Abonent следует выполнить следующий запрос:

```
GRANT UPDATE (Fio, AccountCD) ON Abonent TO PUBLIC;
```

Предложение WITH GRANT OPTION используется для передачи пользователю права передавать привилегии другим пользователям. Например, чтобы передать пользователю Ivanov привилегию чтения таблицы Abonent с правом передавать эту привилегию другим пользователям, необходимо использовать следующий запрос:

```
GRANT SELECT ON TABLE Abonent TO Ivanov  
WITH GRANT OPTION;
```

Для передачи привилегий *нескольким пользователям* имена пользователей должны быть перечислены через запятую. Например, чтобы передать пользователям Ivanov и Petrov все привилегии на таблицу Abonent, необходимо использовать следующий запрос:

```
GRANT ALL ON Abonent TO Ivanov, Petrov;
```

Иногда требуется передача привилегий *хранимым процедурам или триггерам* для доступа к таблицам или представлениям, которые имеют другого владельца. Например, чтобы передать хранимой процедуре Factorial и триггеру Abonent\_DELETE все привилегии доступа на таблицу Abonent, необходимо использовать следующий запрос:

```
GRANT ALL ON Abonent  
TO PROCEDURE Factorial, TRIGGER Abonent_DELETE;
```

Можно задавать привилегии и на выполнение хранимых процедур. Для выполнения (вызова) хранимых процедур пользователями или другими объектами БД необходимо передать EXECUTE привилегию на соответствующую хранимую процедуру.

Рассмотрим пример по передаче привилегий пользователю Ivanov на выполнение хранимой процедуры FactorialSet (которая, в свою очередь, вызывает хранимую процедуру Factorial и записывает результат своей работы в таблицу Ftable). Также будем считать, что *никаких других* привилегий пользователь Ivanov в БД не имеет. Рассмотрим последовательность запросов

GRANT, которые необходимо выполнить для передачи пользователю Ivanov права на выполнение хранимой процедуры FactorialSet:

```
GRANT EXECUTE ON PROCEDURE FactorialSet TO Ivanov;
```

Если пользователь Ivanov попытается выполнить хранимую процедуру FactorialSet, то эта попытка потерпит неудачу, т.к. эта процедура вызывает процедуру Factorial. Таким образом, процедуре FactorialSet требуется привилегия на выполнение хранимой процедуры Factorial, которую SYSDBA может предоставить, используя следующий запрос:

```
GRANT EXECUTE ON PROCEDURE Factorial  
TO PROCEDURE FactorialSet;.
```

Но и в этом случае пользователь Ivanov не сможет выполнить процедуру FactorialSet, т.к. процедура Factorial использует рекурсивный вызов самой себя и при этом производится проверка на наличие привилегий доступа при вызове процедуры Factorial. Таким образом, процедуре Factorial требуется привилегия EXECUTE на процедуру Factorial:

```
GRANT EXECUTE ON PROCEDURE Factorial  
TO PROCEDURE Factorial;.
```

Теперь осталась одна проблема, не позволяющая пользователю Ivanov выполнить процедуру FactorialSet. Дело в том, что процедура FactorialSet (созданная SYSDBA) не имеет доступа на вставку в таблицу Ftable. Следующий запрос GRANT предоставляет необходимую привилегию процедуре FactorialSet:

```
GRANT INSERT ON Ftable TO PROCEDURE FactorialSet;.
```

Таким образом, был рассмотрен один вариант передачи привилегий. Можно также перечисленные выше привилегии передать непосредственно пользователю Ivanov следующим образом:

```
GRANT INSERT ON Ftable TO Ivanov;  
GRANT EXECUTE ON PROCEDURE Factorial TO Ivanov;  
GRANT EXECUTE ON PROCEDURE FactorialSet TO Ivanov;.,
```

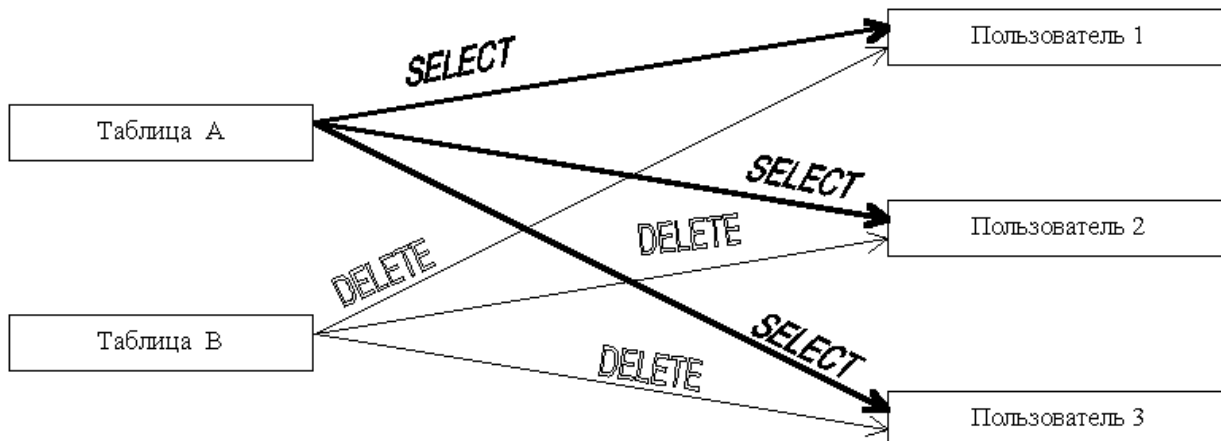
но это является крайне нежелательным с точки зрения безопасности.

Следует отметить, что в IBExpert для передачи привилегий можно использовать «Менеджер прав», который упрощает процедуру наделения различными правами пользователей и других объектов БД, избавляя пользователя от непосредственного написания SQL-запросов.

### 7.1.3. SQL роли

Концепцию планирования безопасности можно представить следующим образом. Администратор SYSDBA в соответствии с информационной моделью создает БД. Затем с помощью запроса GRANT передает привилегии определенным пользователям в соответствии с их уровнем доступа к данным.

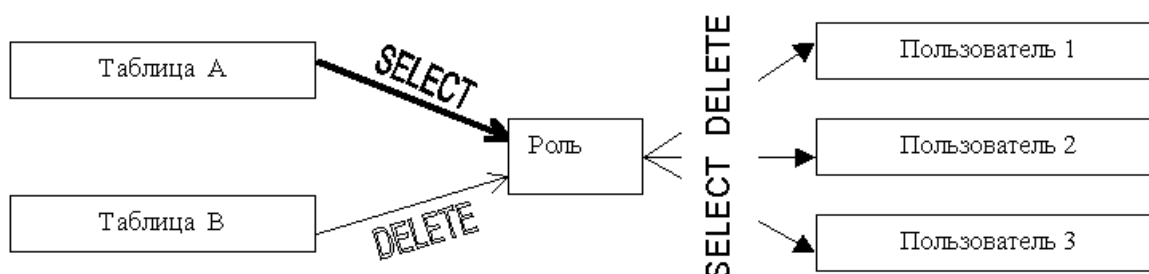
Пусть, например, пользователи 1, 2 и 3 *последовательно* получили привилегию SELECT на таблицу A и привилегию DELETE на таблицу B (рис. 7.1).



**Рис. 7.1.** Пример передачи привилегий

**Роль** представляет собой метод обеспечения безопасности на уровне групп привилегий. Для объединения привилегий `SELECT` и `DELETE` в данном примере используем понятие роли. На рис. 7.2 представлена иллюстрация предыдущего примера, но при использовании механизма роли.

Можно сказать, что роль выступает в качестве объекта БД, концентрирующего в себе заданные привилегии доступа к определенным объектам БД.



**Рис. 7.2.** Пример передачи привилегий при использовании роли

Роль создается запросом `CREATE ROLE`, а удаляется запросом `DROP ROLE`. Эти запросы имеют следующий синтаксис:

```
CREATE ROLE имя_роли;
DROP ROLE имя_роли;
```

Реализация механизма использования роли для передачи привилегий состоит из следующих четырех шагов:

- 1) создать роль, используя следующий запрос:  
`CREATE ROLE имя_роли;`



2) передать этой роли одну или более привилегий, используя следующий запрос:

```
GRANT <привилегии> TO имя_роли;
```

3) передать право на использование этой роли одному или более пользователям с помощью следующего запроса:

```
GRANT имя_роли TO <список_пользователей>;
```

При этом роль может быть передана с указанием предложения WITH ADMIN OPTION, которое означает, что пользователи, обладающие данной ролью, могут передавать право на использование этой роли другим пользователям;

4) при подключении к БД пользователь, обладающий какой-либо ролью (ролями), должен указать имя этой роли.

Например, чтобы передать пользователю Petrov привилегии SELECT, INSERT и DELETE соответственно на таблицы Abonent, Request и PaySumma, используя роль Petrov\_ROLE, объединяющую в себе перечисленные выше привилегии доступа, необходимо применить следующие запросы:

```
CREATE ROLE Petrov_ROLE;  
GRANT SELECT ON Abonent TO Petrov_ROLE;  
GRANT INSERT ON Request TO Petrov_ROLE;  
GRANT DELETE ON PaySumma TO Petrov_ROLE;  
GRANT Petrov_ROLE TO Petrov;
```

Для передачи пользователю Petrov права на передачу роли Petrov\_ROLE другим пользователям необходимо использовать следующий запрос:

```
GRANT Petrov_ROLE TO Petrov WITH ADMIN OPTION;
```

Существуют контекстные переменные для получения имени пользователя и имени роли, под которыми текущий пользователь подключился к БД [18]. Это переменные CURRENT\_USER (тип данных VARCHAR (128), можно использовать переменную USER, описанную ранее) и CURRENT\_ROLE (тип данных VARCHAR (31)). Переменная CURRENT\_ROLE возвращает пустую строку, если текущее соединение не использовало роль.

#### 7.1.4. Отмена привилегий

Для отмены привилегий используется запрос REVOKE, имеющий следующий синтаксис:

```
REVOKE {[GRANT OPTION FOR] <привилегии>  
ON [TABLE] {базовая_таблица | представление}  
FROM {<объект> | <список_пользователей> } }  
| EXECUTE ON PROCEDURE имя_процедуры  
FROM {<объект> | <список_пользователей>}  
| [ADMIN OPTION FOR] <список_ролей>  
FROM {PUBLIC | <список_пользователей>;
```

где <привилегии>, <объект>, <список\_пользователей> и <список\_ролей> имеют такой же синтаксис, как и в запросе GRANT.

При отмене привилегий запросом REVOKE необходимо учитывать следующие ограничения и правила:

- привилегии могут быть удалены только тем пользователем, который их предоставил;
- привилегии, присвоенные другими пользователями, не затрагиваются;
- удаление конкретной привилегии для пользователя А, которому было дано право передавать ее, автоматически удаляет эту привилегию для всех пользователей, кому она была последовательно передана пользователем А;
- привилегии, переданные для общего пользования (PUBLIC), могут быть отобраны только у PUBLIC, а не у конкретного пользователя или объекта БД.

Зарезервированное предложение GRANT OPTION FOR используется для отмены права передавать привилегии другим пользователям. Например, для отмены права пользователя Ivanov передавать привилегию на выборку данных из таблицы Abonent другим пользователям нужно использовать следующий запрос:

```
REVOKE GRANT OPTION FOR SELECT
ON Abonent FROM Ivanov;.
```

После выполнения этого запроса пользователь Ivanov лишится права передавать другим пользователям привилегию SELECT на таблицу Abonent при сохранении им привилегии на доступ к таблице Abonent.

Зарезервированное предложение ADMIN OPTION FOR используется для отмены права передавать роли другим пользователям.

Особенность отмены права передавать привилегии или роли заключается в каскадности такой отмены. Проиллюстрируем это на следующем примере. Пусть на сервере был зарегистрирован пользователь Slonov и администратор SYSDBA передал ему следующие привилегии:

```
GRANT ALL ON PaySumma TO Slonov WITH GRANT OPTION;.
```

Пользователь Slonov выполнил следующий запрос:

```
GRANT SELECT, DELETE ON PaySumma TO Ivanov
WITH GRANT OPTION;.
```

Пользователь Ivanov в свою очередь передал привилегию SELECT пользователю Petrov с помощью следующего запроса:

```
GRANT SELECT ON PaySumma TO Petrov;.
```

Затем администратор SYSDBA отменил право пользователя Slonov передавать привилегию SELECT другим пользователям (с сохранением права передавать DELETE привилегию) с помощью следующего запроса:

```
REVOKE GRANT OPTION FOR SELECT ON PaySumma
FROM Slonov;.
```

Таким образом, в результате каскадного удаления прав:

- пользователь Slonov не имеет права передавать привилегию SELECT;
- пользователь Ivanov вообще лишен привилегии SELECT, а имеет лишь привилегию DELETE (но с опцией WITH GRANT OPTION);

- у пользователя Petrov нет никаких привилегий на доступ к таблице PaySumma. Это произошло, потому что у пользователя Slonov отобрано право передачи привилегии SELECT другим пользователям, а значит, и все действия, которые он произвел, имея это право, также аннулированы (отобрана привилегия SELECT у пользователя Petrov, которую передал ему Ivanov).

Следует отметить, что при передаче пользователям права передавать привилегии может возникнуть проблема множественной передачи. Пусть, например, *каждый* из пользователей – Slonov и Ivanov, имея соответствующие привилегии, выполнили следующий запрос:

```
GRANT SELECT ON Request TO Petrov WITH GRANT OPTION;
```

Потом IVANOV выполнил следующий запрос:

```
REVOKE SELECT ON Request FROM Petrov;
```

и решил, что после этого пользователь Petrov не имеет доступа к таблице Request. Но у пользователя Petrov сохранилась привилегия, которую передал ему Slonov, и он по-прежнему имеет доступ к таблице Request.

Таким образом, передача привилегий с правом последующей их передачи должна использоваться с большой осторожностью или не использоваться вовсе.

Чтобы удалить привилегии, переданные какой-либо роли, можно использовать запрос REVOKE или просто удалить соответствующую роль со всеми ее привилегиями. Например, удаление роли Petrov\_ROLE будет выглядеть следующим образом:

```
DROP ROLE Petrov_ROLE;
```

### 7.1.5. Привилегии на представления

Для контроля доступа к таблицам БД могут быть использованы представления. Представление обычно создается как подмножество столбцов и строк для одной или нескольких таблиц. Из-за этого представление в определенной степени обеспечивает безопасность доступа. Для обеспечения доступа представлений к базовым объектам и последующего доступа пользователей к представлению необходимо использовать запрос GRANT.

При создании представления только для чтения создателю нужны привилегии SELECT к каждой базовой таблице, используемой представлением. Для определения обновляемых представлений создателю нужны привилегии ALL к соответствующим базовым таблицам.

Пусть, например администратор SYSDBA передал привилегии пользователю Petrov с помощью следующего запроса:

```
GRANT ALL ON Abonent TO Petrov;
```

Допустим, что Petrov создает представление Ab\_fio, используя следующий запрос:

```
CREATE VIEW Ab_fio AS SELECT AccountCD, Fio FROM Abonent;
```

Данный запрос будет выполнен успешно, так как Petrov имеет права на доступ к таблице Abonent. Допустим, Petrov пытается создать еще одно представление и формирует следующий запрос:

```
CREATE VIEW Req  
AS SELECT AccountCD, IncomingDate FROM Request;
```

Данный запрос не будет выполнен и будет выдано сообщение об ошибке "no permission for SELECT access to TABLE/VIEW REQUEST" (нет разрешения на SELECT доступ к таблице Request).

Для обеспечения доступа к представлению только для чтения владелец должен предоставить пользователям привилегию SELECT на это представление.

**Примечание.** Для самого представления не требуется передавать привилегии на доступ к таблицам, на которых оно основано.

На обновляемые представления могут передаваться привилегии INSERT, DELETE и UPDATE. Однако существуют определенные сложности, если представление является обновляемым или если оно включает другие представления или ХП выбора. Если представление обновляемое, то модификации данных в таком представлении приводят к модификациям данных в базовых таблицах. Поэтому владельцы базовых таблиц должны предоставить пользователю соответствующие права доступа (INSERT, DELETE, UPDATE) к этим таблицам. Если представление обращается к другим представлениям или ХП выбора, то их владельцы должны предоставить пользователю привилегию SELECT на соответствующую ХП или представление. Привилегии к базовым объектам также должны быть предоставлены *самому представлению*.

Привилегии REFERENCES неприменимы к представлениям за исключением следующей ситуации. Если представление использует таблицу, которая имеет внешние ключи, ссылающиеся на другие таблицы, то представлению нужны привилегии REFERENCES к этим другим таблицам, если эти таблицы сами не используются в данном представлении.

Например, пусть пользователь Petrov создает представление с помощью следующего запроса:

```
CREATE VIEW Ab_Str  
AS SELECT AccountCD, StreetCD, Fio FROM Abonent;
```

Затем Petrov пробует обновить строку в представлении следующим образом:

```
UPDATE Ab_Str SET StreetCD=7  
WHERE AccountCD='005488';
```

Этот запрос не будет выполнен, пока не будут переданы привилегии с помощью следующего запроса:

```
GRANT REFERENCES (StreetCD) ON Street TO Ab_Str;
```

Причем в данном случае, так как Petrov не имеет прав доступа к таблице Street, последний оператор может быть выполнен только администратором.

## 7.2. Транзакции

### 7.2.1. Понятие транзакции

Область организации транзакций и управления ими очень широка. Мы ограничимся рассмотрением наиболее общих вопросов управления транзакциями в пределах языка SQL.

Транзакция – это группа операций обработки данных, выполняемых как некоторое неделимое действие над базой данных, осмысленное с точки зрения пользователя. Запись данных в БД производится только при успешном выполнении всех операций группы. Если хотя бы одна из операций группы завершается неуспешно, то БД возвращается к тому состоянию, в котором она была до выполнения первой операции группы. Транзакция реализует некоторую прикладную функцию, например перевод денег с одного счета на другой при оплате услуг по безналичному расчету (снять и положить). Операция перевода денег с банковского счета абонента, производящего оплату безналичным путем, на счет организации, предоставившей услуги газоснабжения, должна составлять единую транзакцию. Иначе может возникнуть ситуация, когда первый SQL-оператор переведет деньги на другой счет, а второй, выполняющий снятие их со счета, не доведет дело до конца из-за непредвиденного сбоя.

Транзакции характеризуются четырьмя классическими свойствами: атомарности, согласованности, изолированности, долговечности (прочности) - ACID (Atomicity, Consistency, Isolation, Durability) [1]. Поэтому часто транзакции называют ACID-транзакциями. Эти свойства означают следующее.

1. *Свойство атомарности* выражается в том, что транзакция должна быть выполнена в целом или не выполнена вовсе.  
СУБД гарантирует невозможность фиксации некоторой части действий из транзакции в БД.
2. *Свойство согласованности* гарантирует, что по мере выполнения транзакций данные переходят из одного согласованного состояния в другое, т.е. транзакция не разрушает взаимной согласованности данных.
3. *Свойство изолированности* означает, что конкурирующие за доступ к базе данных транзакции физически обрабатываются последовательно, изолированно друг от друга, но для пользователей это выглядит так, как будто они выполняются параллельно. Например, для любых двух транзакций T1 и T2 справедливо следующее утверждение: T1 сможет увидеть обновление T2 только после выполнения T2, а T2 сможет увидеть обновление T1 только после выполнения T1.
4. *Свойство долговечности* означает, что если транзакция завершена успешно, то изменения в данных, произведенные в ней, не могут быть потеряны ни при каких обстоятельствах (даже в случае последующих ошибок или сбоя системы).

Таким образом, использование транзакций имеет следующие преимущества.

1. Механизм транзакций позволяет обеспечить логическую целостность данных в БД. Другими словами, транзакции – это логические единицы работы, после выполнения которых БД остается в целостном состоянии.
2. Транзакции также являются единицами восстановления данных. Восстанавливаясь после сбоев, система ликвидирует следы транзакций, не успевших успешно завершиться в результате программного или аппаратного сбоя.
3. Механизм транзакций обеспечивает правильность работы в многопользовательских системах при параллельном обращении нескольких пользователей к одним и тем же данным.

Рассмотрим подробнее особенности языка SQL для управления транзакциями.

### 7.2.2. Восстановление данных

Восстановление в СУБД означает способность возвращения базы данных в правильное состояние, если какой-то сбой сделал текущее состояние неправильным или подозрительным. Основной принцип восстановления – избыточность. Избыточность обеспечивается фиксацией транзакций, а восстановление – откатом транзакций.

Системный компонент СУБД, обеспечивающий ACID-свойства транзакций, называется *администратором транзакций*. Он содержит команды COMMIT и ROLLBACK.

Команда COMMIT завершает текущую транзакцию, выполняя фиксацию сделанных изменений в базе данных. Иногда говорят, что команда COMMIT фиксирует транзакцию.

*Фиксация транзакции* – это действие, обеспечивающее запись на диск изменений в базе данных, сделанных при выполнении транзакции. До тех пор, пока транзакция не зафиксирована, существует возможность аннулирования этих изменений, восстановления базы данных в то состояние, в котором она была на момент начала транзакции. Таким образом, фиксация текущей транзакции означает следующее:

- все результаты выполнения транзакции становятся постоянными;
- результаты будут видны другим транзакциям (до этого момента все данные, затрагиваемые транзакцией, будут "видны" пользователю в состоянии на начало текущей транзакции).

Команда COMMIT имеет следующий синтаксис:

COMMIT [WORK] [RETAIN];

Слово WORK является необязательным, поэтому команда COMMIT WORK полностью аналогична команде COMMIT. Команда COMMIT (иногда называемая *жестким подтверждением*) освобождает все физические ресурсы,

связанные с транзакцией. Использование ключевого слова RETAIN после команды COMMIT означает, что транзакция зафиксирована, но физические ресурсы не будут освобождены. Такое подтверждение иногда называют *мягким подтверждением*. Оно может быть полезным при выполнении логической задачи, которая включает в себя множество повторений похожих операций. Например, мягкое подтверждение сохраняет открытые в настоящий момент курсоры для выбранных наборов.

Команда ROLLBACK выполняет откат транзакции. Откат транзакции – это действие, обеспечивающее аннулирование всех изменений данных, которые были сделаны операторами SQL в теле текущей незавершенной транзакции. Каждый оператор в транзакции выполняет свою часть работы, но для успешного завершения всей работы в целом требуется безусловное завершение всех их операторов.

Команда ROLLBACK имеет следующий синтаксис:

```
ROLLBACK [WORK] [RETAIN] [TO [SAVEPOINT] имя_точки_сохранения];
```

Как и COMMIT, команда ROLLBACK освобождает ресурсы на сервере. При использовании ROLLBACK RETAIN выделенные ресурсы не освобождаются, курсоры сохраняются. Такой откат следует использовать с большой осторожностью, поскольку отдельные вызовы откатов производятся в ответ на исключение некоторого вида. В таком случае использование ROLLBACK RETAIN также сохранит и причины исключения.

Новая транзакция начинается с начала каждого сеанса работы с базой данных. Далее все выполняемые SQL-операторы будут входить в одну транзакцию до тех пор, пока не будет выполнена команда COMMIT или ROLLBACK. Т.е. чтобы начать транзакцию, не требуется выполнять никаких специальных действий. Транзакция начинается автоматически вместе с первым SQL-оператором или непосредственно после окончания предыдущей транзакции.

Логически транзакция должна объединять только выполнение взаимосвязанных операций. Так, если делать транзакции "очень большими", состоящими из длинной последовательности не связанных между собой операторов, то любой сбой, автоматически выполняющий откат транзакции, повлияет на отмену действий, которые могли бы быть успешно завершены при более "коротких" транзакциях.

Рассмотрим транзакцию "Изменить оплату с кодом 6 с 20 на 100, а затем изменить дату оплаты с '06/13/2001' на '06/11/2001'". Предполагается, что пользователь монопольно вводит запросы SQL последовательно один за другим, пользуясь некоторым интерактивным средством (например, IBExpert). Тело транзакции будут составлять следующие запросы:

```
UPDATE PaySumma SET PaySum = 100
WHERE PayFactCD = 6;
UPDATE PaySumma SET PayDate = '06/11/2001'
WHERE PayFactCD = 6;
```

*...К этому моменту никаких ошибок не обнаружено...*

**COMMIT WORK;**

Последняя команда фиксирует изменения в базе данных, достигнутые в результате выполнения запросов SQL, составляющих транзакцию.

Вновь рассмотрим ту же самую транзакцию, но предположим, что в процессе ввода запросов SQL пользователь допустил ошибку, указав в последнем запросе ошибочную дату, например:

```
UPDATE PaySumma SET PaySum = 100
WHERE PayFactCD = 6;
UPDATE PaySumma SET PayDate = '05/11/2001'
WHERE PayFactCD = 6;
```

Так как произошла ошибка, используется команда ROLLBACK WORK. Все изменения в базе данных, декларированные запросами внутри транзакции, отменяются, и база данных возвращается к состоянию на момент начала транзакции.

Таким образом, возможны два варианта завершения транзакции:

- транзакция фиксируется, если все операторы выполнены успешно и в процессе выполнения транзакции не произошло никаких сбоев программного или аппаратного обеспечения;
- база данных должна быть возвращена в исходное состояние, если в процессе выполнения транзакции случилось то, что делает невозможным ее нормальное завершение.

В общем случае транзакция завершается одним из четырех возможных путей:

- использование команды COMMIT в случае успешного завершения транзакции;
- использование команды ROLLBACK в случае прерывания транзакции;
- успешное завершение программы, в которой была инициирована текущая транзакция, означает успешное завершение транзакции (как будто была использована команда COMMIT);
- ошибочное завершение программы прерывает транзакцию (как будто была использована команда ROLLBACK).

На рис. 7.3 в графическом виде представлены типичные транзакции, иллюстрирующие четыре перечисленные ситуации [11].



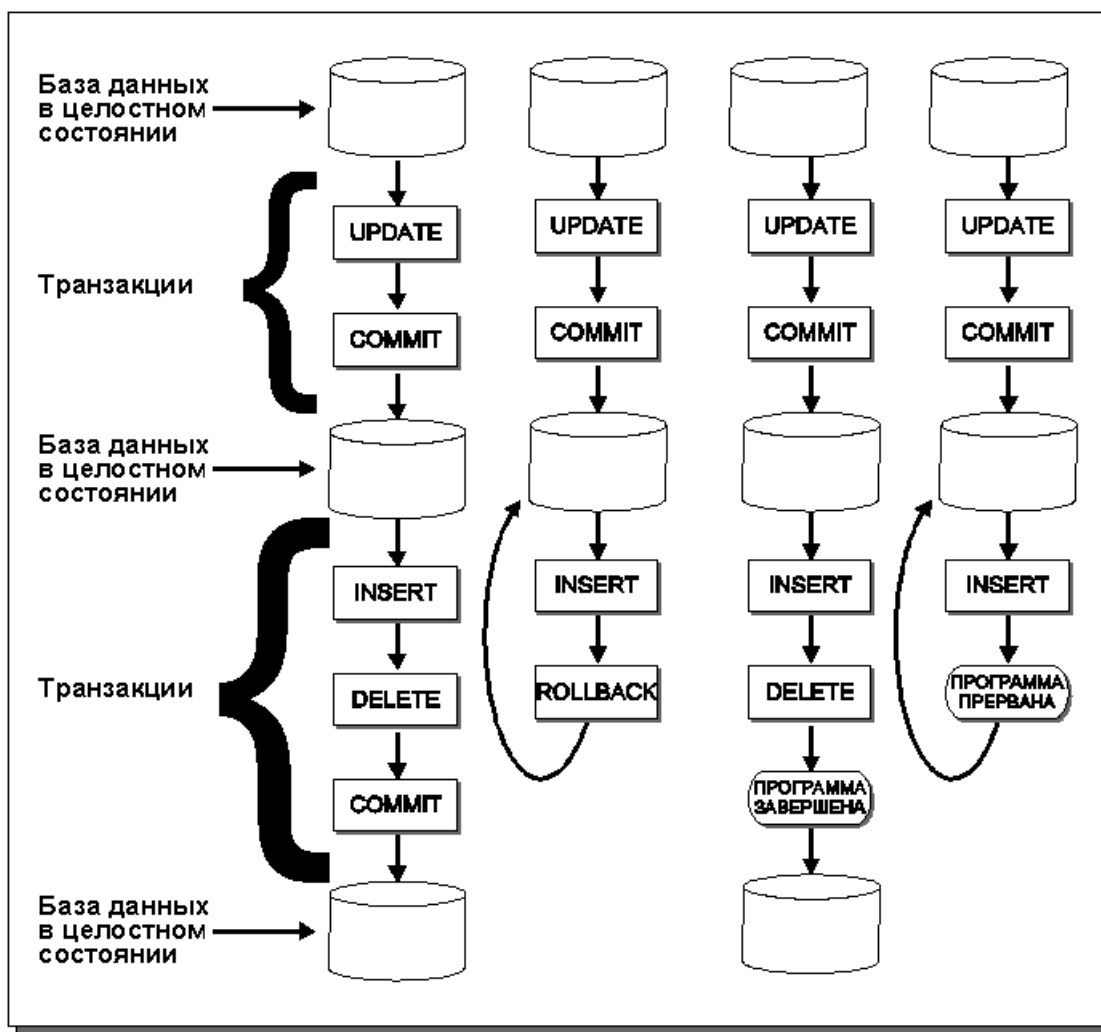


Рис. 7.3. Выполненные и отмененные транзакции

В СУБД Firebird существует возможность задавать *пользовательские точки сохранения* (альтернативное имя – *вложенные транзакции*) и затем откатывать транзакцию не полностью, а частично – до этих точек [18, 28]. Место расположения точки сохранения в транзакции указывает оператор SAVEPOINT, который имеет следующий синтаксис:

SAVEPOINT имя\_точки\_сохранения;.

Правило составления имя\_точки\_сохранения такое же, как и для любых других идентификаторов языка SQL (см. п. 2.7).

В транзакции может быть определено множество точек сохранения, причем если в одной транзакции повторно определяется точка с уже использовавшимся именем, то она перекрывает существовавшую точку, связанную с этим именем.

Возврат на точку сохранения начинается с отмены всей работы, выполненной после создания этой точки сохранения. Указанная точка сохранения и все предшествующие ей сохраняются, а те точки, которые были созданы после указанной точки сохранения, пропадают. Еще одна особенность отката до

контрольной точки заключается в том, что транзакция не завершается. Т.е. оставшиеся изменения, если требуется, надо зафиксировать.

Рассмотрим следующий пример использования точек сохранения:

```
CREATE TABLE Test (ID INTEGER);
COMMIT;
INSERT INTO Test VALUES (1);
COMMIT;
INSERT INTO Test VALUES (2);
SAVEPOINT y;
DELETE FROM Test;
SELECT * FROM Test; -- не вернет ничего
ROLLBACK TO y;
SELECT * FROM Test; -- вернет две строки
ROLLBACK;
SELECT * FROM Test; -- вернет одну строку.
```

Механизм реализации точек сохранения на сервере может требовать значительного количества ресурсов, особенно если одни и те же строки изменяются многократно в течение одной транзакции. Ресурсы уже ненужных точек сохранения могут быть освобождены при использовании оператора `RELEASE SAVEPOINTS`, имеющего следующий синтаксис:

```
RELEASE SAVEPOINTS имя_точки_сохранения [ONLY];
```

Без ключевого слова `ONLY` указанная точка сохранения и все точки сохранения, которые были созданы после нее, будут освобождены и потеряны. `ONLY` используется для освобождения только указанной точки сохранения.

Точки сохранения оказываются полезными в интерактивных программах, так как они позволяют выделять и именовать промежуточные шаги программы [16]. Это позволяет гибко управлять сложными программами. Например, можно проставить точки сохранения в длинной цепочке обновлений. При этом, обнаружив ошибки, можно будет произвести откат к нужному месту и тем самым не выполнять повторно все обновления, как если бы был произведен откат всей транзакции.

Точки сохранения также оказываются полезными при написании прикладных программ. Если программа содержит подпрограммы, можно проставить точки сохранения перед обращением к каждой из подпрограмм. Если подпрограмма завершилась аварийно, можно восстановить БД на момент до выполнения этой подпрограммы.

### 7.2.3. Восстановление системы

Система должна быть готова к восстановлению не только после небольших локальных нарушений (таких, как невыполнение операции в пределах

определенной транзакции), но также и после глобальных нарушений (типа сбоев питания).

Местное нарушение поражает только транзакцию, в которой оно произошло.

Глобальное нарушение поражает сразу все транзакции и приводит к значительным для системы последствиям. Возможны два вида глобальных нарушений:

- отказы системы (например, из-за питания), поражающие все выполняющиеся в данный момент транзакции, но физически не нарушающие базу данных в целом. Эти отказы называют аварийным отказом программного обеспечения.
- отказы носителей (например, поломка головок дискового накопителя), которые могут представлять угрозу для базы данных или какой-либо ее части и поражать, по крайней мере, те транзакции, которые используют эту часть базы данных. Эти отказы называют аварийным отказом аппаратного обеспечения.

Для восстановления системы применяются различные методы, основанные на использовании **журнала транзакций** [5, 24].

Когда пользователь выполняет запрос на изменение базы данных, СУБД автоматически вносит в журнал транзакций одну запись для каждой строки, измененной данным оператором. Эта запись содержит две копии строки. Одна копия представляет собой строку до изменения, а другая - после изменения. Только после того как в журнале будет сделана запись, СУБД изменит физическую строку на диске. Затем, если пользователь выполнит оператор COMMIT, в журнале отмечается конец транзакции. При выполнении ROLLBACK СУБД обращается к журналу и извлекает из него «исходные» копии строк, измененные во время транзакции.

Используя эти копии, СУБД возвращает строки в прежнее состояние и таким образом отменяет изменения, внесенные в БД во время выполнения транзакции.

В случае системного сбоя механизм восстановления системы БД должен выяснить два вопроса:

- какие транзакции не успели завершиться к моменту сбоя и, следовательно, должны быть отменены;
- какие транзакции успели завершиться к моменту сбоя, но соответствующая информация еще не переписалась из внутренних буферов системы в саму БД (физическую), следовательно, должны быть выполнены повторно.

Администратор БД восстанавливает ее с помощью специальной утилиты восстановления, поставляемой вместе с СУБД (для СУБД Firebird это утилита командной строки gfix.exe).

С целью сокращения времени обработки в современных СУБД журнал транзакций обычно хранится на отдельном более скоростном жестком диске. Имеется также возможность отключать ведение журнала транзакций.

БД могут быть локальными, сетевыми и распределенными.

Если данные хранятся в одной базе данных, то транзакция к ней рассматривается как *локальная*. Все рассмотренное выше применимо к локальным БД.

Если с БД одновременно работают несколько пользователей, то обработка транзакций приобретает новое измерение (параллелизм, который будет рассмотрен позже).

Распределенные системы обычно включают несколько компьютеров – серверов баз данных, называемых **узлами**. Данные физически распределены между ними. На каждом узле содержится некоторая локальная база данных, содержащая фрагмент данных из общей распределенной базы. В распределенных базах транзакция, выполнение которой заключается в обновлении данных на нескольких узлах сети, называется глобальной или распределенной транзакцией.

Внешне выполнение распределенной транзакции выглядит как обработка транзакции к локальной базе данных. Тем не менее, распределенная транзакция включает в себя несколько локальных транзакций, каждая из которых завершается двумя путями – фиксируется или прерывается. Распределенная транзакция фиксируется только в том случае, когда зафиксированы все локальные транзакции, ее составляющие. Если хотя бы одна из локальных транзакций была прервана, то должна быть прервана и распределенная транзакция.

Для обработки распределенных транзакций в современных СУБД предусмотрен так называемый **протокол двухфазной фиксации транзакций (two-phase commit)**. Т.е. фиксация распределенной транзакции выполняется в две фазы [1, 11].

**Фаза 1** начинается, когда при обработке транзакции встретился оператор СОММИТ. Сервер распределенной БД (или компонент СУБД, отвечающий за обработку распределенных транзакций) направляет уведомление "подготовиться к фиксации" всем серверам локальных БД, выполняющим распределенную транзакцию. Если все серверы приготовились к фиксации (то есть откликнулись на уведомление, и их отклик был получен), сервер распределенной БД принимает решение о фиксации. Серверы локальных БД остаются в состоянии готовности и ожидают от него команды "зафиксировать". Если хотя бы один из серверов не откликнулся на уведомление в силу каких-либо причин, будь то аппаратная или программная ошибка, то сервер распределенной БД откатывает локальные транзакции на всех узлах, включая даже те, которые подготовились к фиксации и оповестили его об этом.

**Фаза 2** – сервер распределенной БД направляет команду "зафиксировать" всем узлам, затронутым транзакцией, и гарантирует, что транзакции на них будут зафиксированы. Если связь с локальной базой данных потеряна в интервал времени между моментом, когда сервер распределенной БД принимает решение о фиксации транзакции, и моментом, когда сервер локальной БД подчиняется его команде, то сервер распределенной БД продолжает попытки завершить транзакцию, пока связь не будет восстановлена.

## 7.2.4. Параллелизм

Одной из основных задач многопользовательских СУБД является организация одновременного доступа к одним и тем же данным множеству пользователей с помощью механизма транзакций. Таким образом, когда две или более задач выполняются над одной и той же БД в одно и то же время, говорят о *параллелизме*. Основными проблемами, возникающими при параллельной обработке транзакций, являются следующие [5]:

- потерянные или "скрытые" обновления;
- зависимость от незафиксированных данных ("грязное" чтение);
- несогласованный анализ (неповторяемое чтение);
- чтение фантомов.

**Потерянные обновления** появляются в ситуации, когда две или более программ читают одни и те же данные из базы данных, вносят в них какие-либо изменения и затем пытаются одновременно записать результат на прежнем месте. В этом случае ни у одной из транзакций нет сведений о действиях, выполненных другими транзакциями. В базе данных могут быть сохранены изменения, выполненные только *одной программой*, – любые другие изменения будут потеряны.

**Зависимость от незафиксированных данных** возникает, когда вторая транзакция выбирает строку, которую в это время обновляет первая транзакция. Затем первая транзакция откатывается оператором ROLLBACK. Таким образом, вторая транзакция читает еще не зафиксированные данные, которые могут быть изменены первой транзакцией.

**Несогласованный анализ** возникает тогда, когда одна программа получает какой-то итоговый отчет, а в это время другая программа изменяет данные, используемые первой программой. Несогласованный анализ напоминает неподтвержденную зависимость тем, что первая транзакция изменяет данные, которые читает вторая транзакция. Однако при несогласованном анализе первая транзакция подтверждает изменение этих данных. Кроме того, при несогласованном анализе происходит многократное чтение второй транзакцией одной и той же строки с разными данными.

**Чтение фантомов** возникает, когда последующий набор строк, прочитанный транзакцией, отличается от набора, который был прочитан в начале работы транзакции. Фантомные строки появляются, если при последующем чтении появляются новые добавленные строки и/или исчезают удаленные строки, которые были подтверждены с момента первого чтения.

Поэтому необходим определенный механизм обработки транзакций, позволяющий устранить подобные проблемы. Такой механизм существует и опирается на следующие правила [1, 11].

1. В процессе выполнения транзакции пользователь (или программа) "видит" только согласованные состояния базы данных. Пользователь (или программа) никогда не может получить доступ к незафиксированным обновлениям в данных, достигнутым в результате действий другого пользователя (программы).

2. Если две транзакции, А и В, выполняются параллельно, то СУБД полагает, что результат будет такой же, как если бы:

- транзакция А выполнялась первой, за ней была бы выполнена транзакция В;
- транзакция В выполнялась бы первой, за ней была бы выполнена транзакция А.

Администратор транзакций гарантирует, что каждый пользователь (программа), обращающийся к базе данных, работает с ней так, как будто не существует других пользователей (программ), одновременно с ним обращающихся к тем же данным. Для практической реализации этого СУБД используют **механизм блокировок** [1, 11]. В случае, когда для выполнения некоторой транзакции необходимо, чтобы некоторый объект не изменялся непредсказуемо и без ведома этой транзакции, такой объект блокируется.

Таким образом, механизм блокировок разрешает проблемы, связанные с доступом нескольких пользователей (программ) к одним и тем же данным. Однако его применение связано с существенным замедлением обработки транзакций, вызванным необходимостью ожидания освобождения данных, захваченных конкурирующей транзакцией.

Возможно ускорение обработки путем локализации фрагментов данных, захватываемых транзакцией.

СУБД может блокировать:

- всю базу данных целиком;
- таблицу базы данных;
- часть таблицы;
- отдельную строку.

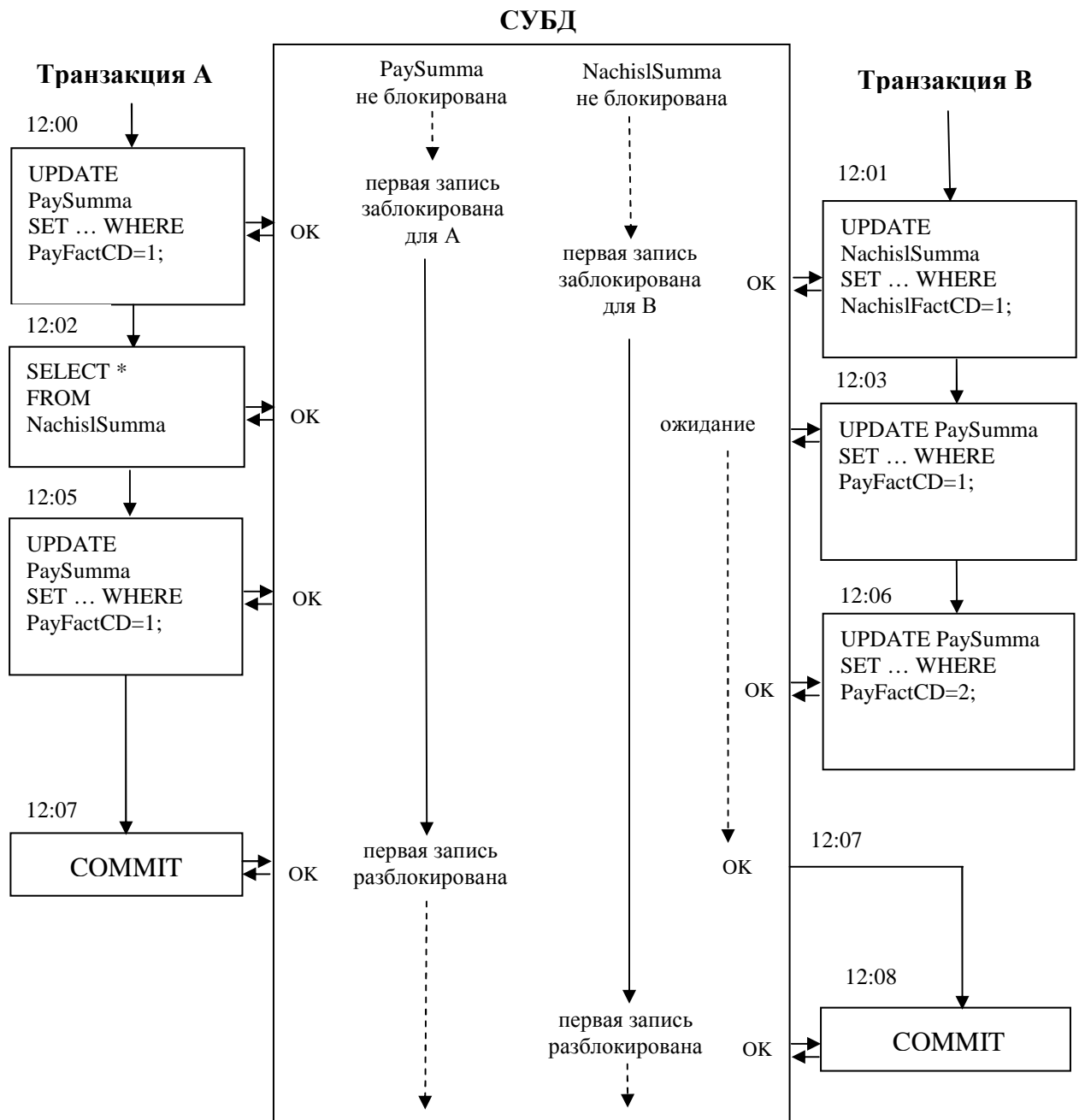
Это называется **уровнями блокировки**. Современные СУБД в основном используют блокировки на уровне частей таблиц (страниц) и/или на уровне записей.

При блокировке на уровне страниц СУБД захватывает для выполнения транзакции некоторый фрагмент таблицы, запрещая доступ к нему (на время выполнения транзакции) конкурирующим транзакциям. Последние, впрочем, могут захватить другие страницы той же таблицы. Так как размер страниц обычно невелик (1-8 Кб), то время ожидания транзакций, конкурирующих за доступ к страницам таблицы, оказывается приемлемым даже для режима оперативного доступа к базе данных.

Если СУБД имеет возможность блокировать для выполнения транзакции отдельные строки таблицы, то скорость обработки транзакции существенно повышается. Блокировка на уровне записей (строк) позволяет добиться максимальной производительности за счет того, что захватываемый объект (запись) является минимальной структурной единицей базы данных. Блокировка на уровне записей (row-level) реализована в Firebird.

Теоретически блокировка на уровне элементов данных (захват конкретного поля строки) позволит добиться еще большей производительности.

Для пояснения действия механизма блокировок на уровне записей, реализованного в СУБД Firebird, воспользуемся рис. 7.4.



**Рис. 7.4.** Использование блокировок на уровне записей

На рис. 7.4 представлены две таблицы учебной базы данных (PaySumma, NachislSumma), к которым возможен доступ в процессе выполнения двух транзакций - А и В. Сначала при выполнении транзакции А происходит обновление первой записи таблицы PaySumma. СУБД блокирует данную запись до тех пор, пока транзакция А не будет зафиксирована или отменена. Транзакция В выполняется параллельно и блокирует на момент своего выполнения первую запись таблицы NachislSumma. Затем в контексте транзакции А происходит выборка из таблицы NachislSumma. Такая выборка возможна (так как транзакцией В заблокирована первая запись таблицы только для обновления, но не для чтения), однако транзакция А не увидит изменений,

сделанных неподтвержденным обновлением в транзакции В. Затем в процессе выполнения транзакции В делается попытка доступа к заблокированной первой записи таблицы PaySumma. В результате обработка транзакции В приостанавливается и возобновляется только после того, как транзакция А завершается и освобождает заблокированную ею запись таблицы PaySumma. Вместе с тем попытка обновления второй записи таблицы PaySumma в процессе выполнения транзакции В будет успешной, так как заблокирована не вся таблица PaySumma, а лишь первая запись.

Помимо уровней блокировки, выделяют также тип блокировки или схему блокировки. Конкурирующие транзакции могут захватывать данные, в то же время, разрешая доступ к этим данным другим транзакциям, но только для чтения. Кроме того, транзакции могут блокировать данные, не допуская захвата тех же данных другими транзакциями, в том числе и только для чтения.

## **Контрольные вопросы**

1. Что может быть причиной разрушения или потери данных в БД?
2. Какие общие требования предъявляются к безопасности реляционных СУБД? На каких принципах базируется схема доступа к данным?
3. Какие привилегии доступа существуют в БД?
4. Как выполнить передачу привилегий с помощью SQL-запроса?
5. Из каких шагов состоит реализация механизма использования SQL-роли?
6. Каким образом осуществляется отмена привилегий на языке SQL?
7. Какие ограничения и правила необходимо учитывать при отмене привилегий оператором REVOKE?
8. Какие особенности связаны с передачей привилегий при использовании представлений?
9. Что такое транзакция? Какими ACID-свойствами характеризуются транзакции? В чем состоят преимущества использования транзакций?
10. Что такое фиксация транзакции? Как осуществляется фиксация транзакции на языке SQL?
11. Что такое откат транзакции? Как осуществляется откат транзакции на языке SQL?
12. Как можно использовать пользовательские точки сохранения при работе с транзакциями?



## Приложение А

### Описание учебной базы данных

Учебная БД представляет собой очень упрощенный пример информационной модели расчетной компоненты расчетно-платежного комплекса «Абонент+», которая используется для информационного обеспечения деятельности газораспределительных организаций и региональных компаний по реализации газа по оказанию населению услуг газоснабжения [3].

Учебная БД состоит из восьми таблиц: пяти таблиц-справочников и трех информационных таблиц.

В учебной БД используются следующие справочники.

1. Справочник улиц, на которых проживают абоненты (таблица Street).
2. Справочник абонентов (таблица Abonent).
3. Справочник услуг газоснабжения, оказываемых абонентам газовой сети (таблица Services).
4. Справочник возможных неисправностей газового оборудования абонентов (таблица Disrepair).
5. Справочник исполнителей заявок, поданных абонентами на ремонт газового оборудования. Исполнителями являются работники ремонтной службы газораспределительной организации, оказываемой абонентам услуги газоснабжения (таблица Executor).

В качестве информационных таблиц учебной БД выделены следующие таблицы.

1. Таблица NachislSumma для хранения информации о размере ежемесячного начисления абонентам за расходуемый газ или оказание других услуг газоснабжения (которые расшифровываются в справочнике услуг газоснабжения).  
Начисление за ремонт газового оборудования производится по факту оказания услуг.
2. Таблица PaySumma для хранения величин оплаченных сумм за оказанные услуги газоснабжения. Для каждого факта оплаты по какой-либо услуге газоснабжения указывается оплачиваемый месяц и год. Таким образом, при сопоставлении информации по конкретному абоненту, хранящейся в таблице NachislSumma, можно узнать размер долга или переплаты у данного абонента на указанный месяц.
3. Таблица Request для хранения информации о заявках абонентов на ремонт газового оборудования. Каждая ремонтная заявка характеризуется номером лицевого счета заявившего абонента (расшифровка в справочнике Abonent), определенной неисправностью газового оборудования (расшифровка в справочнике Disrepair), исполнителем ремонтной работы (справочник Executor), датой регистрации заявки, датой выполнения ремонта и признаком погашения (1/0).

Ниже приводится описание назначения всех полей для каждой таблицы учебной базы данных.

Назначение полей таблицы Street (справочник улиц):

- StreetCD – уникальный код улицы (первичный ключ таблицы Street);
- StreetNM – название улицы, расшифровывающее код улицы.

Назначение полей таблицы Abonent (справочник абонентов):

- AccountCD – номер лицевого счета абонента, уникальным образом идентифицирующий каждого из абонентов (первичный ключ таблицы Abonent);
- StreetCD – код улицы, на которой проживает абонент (внешний ключ, ссылающийся на первичный ключ таблицы Street);
- HouseNo – номер дома, в котором проживает абонент;
- FlatNo – номер квартиры;
- Fio – фамилия, имя и отчество абонента в формате "Фамилия И.О.";
- Phone – номер телефона.

Назначение полей таблицы Services (справочник услуг газоснабжения):

- GazServiceCD – код услуги газоснабжения (первичный ключ таблицы Services);
- GazServiceNM – наименование услуги газоснабжения.

Назначение полей таблицы Disrepair (справочник наименований неисправностей газового оборудования):

- FailureCD – код неисправности газового оборудования (первичный ключ таблицы Disrepair);
- FailureNM – наименование неисправности газового оборудования.

Назначение полей таблицы Executor (справочник исполнителей ремонтных заявок):

- ExecutorCD – уникальный код, идентифицирующий исполнителей ремонтных заявок (первичный ключ таблицы Executor);
- Fio – фамилия, имя и отчество исполнителя в формате «Фамилия И.О.».

Назначение полей таблицы NachislSumma (хранение сумм ежемесячного начисления):

- NachislFactCD – уникальный идентификатор факта начисления (первичный ключ таблицы NachislSumma);
- AccountCD – номер лицевого счета абонента, которому было сделано начисление (внешний ключ, ссылающийся на первичный ключ таблицы Abonent);
- GazServiceCD – код услуги газоснабжения, за которую выполнено начисление (внешний ключ, ссылающийся на первичный ключ таблицы Services справочника услуг газоснабжения);
- NachislSum – значение начисленной суммы;
- NachislMonth – номер месяца, за который произведено начисление с идентификатором факта начисления, хранящимся в поле NachislFactCD;
- NachislYear – год, за месяц которого выполнено начисление.

Назначение полей таблицы PaySumma (хранение оплаченных сумм):

- PayFactCD – уникальный идентификатор факта оплаты абонентом по услуге газоснабжения (первичный ключ таблицы PaySumma);
- AccountCD – номер лицевого счета абонента, оплатившего оказанную ему услугу газоснабжения (внешний ключ, ссылающийся на первичный ключ таблицы Abonent);
- GazServiceCD – код оплаченной услуги газоснабжения (внешний ключ, ссылающийся на первичный ключ справочника услуг газоснабжения Services);
- PaySum – значение оплаченной суммы;
- PayDate – дата оплаты;
- PayMonth – номер оплачиваемого месяца;
- PayYear – оплачиваемый год.

Назначение полей таблицы Request (хранение ремонтных заявок):

- RequestCD – уникальный код ремонтной заявки (первичный ключ таблицы Request);
- AccountCD – номер лицевого счета абонента, подавшего данную ремонтную заявку (внешний ключ, ссылающийся на первичный ключ таблицы Abonent);
- FailureCD – код неисправности газового оборудования, заявленной абонентом в данной ремонтной заявке (внешний ключ, ссылающийся на первичный ключ таблицы Disrepair);
- ExecutorCD – код исполнителя, ответственного за выполнение данной ремонтной заявки (внешний ключ, ссылающийся на первичный ключ таблицы Executor);
- IncomingDate – дата поступления заявки;
- ExecutionDate – дата выполнения заявки;
- Executed – поле логического типа, признак того, погашена заявка или нет.

Для поддержания правил ссылочной целостности, реализующих запрет удаления записи в родительской таблице при наличии связанных записей в дочерних таблицах, в учебной базе данных определены следующие триггеры.

#### 1. TD\_ABONENT.

Триггер запускается после удаления строки в таблице Abonent. Если в таблицах NachisSumma или PaySumma имеются записи с внешним ключом AccountCD, ссылающимся на удаляемую строку таблицы Abonent, то триггер вызывает исключение Del\_Restrict и операция удаления прерывается.

#### 2. TD\_SERVICES.

Триггер запускается после удаления строки в таблице Services. Если в таблицах PaySumma или NachisSumma имеются записи с внешним ключом GazServiceCD, ссылающимся на удаляемую строку в таблице Services, то триггер вызывает пользовательское исключение Del\_Restrict и прерывает выполнение операции.

Текст определения описанных триггеров приведен в скрипте по созданию учебной БД (приложение Б).

Ниже приводятся данные таблиц базы данных.  
В таблице А.1 приведены данные таблицы Street.

Таблица А.1 – Данные таблицы Street

STREETCD	STREETNM
1	ЦИОЛКОВСКОГО УЛИЦА
2	НОВАЯ УЛИЦА
3	ВОЙКОВ ПЕРЕУЛОК
4	ТАТАРСКАЯ УЛИЦА
5	ГАГАРИНА УЛИЦА
6	МОСКОВСКАЯ УЛИЦА
7	КУТУЗОВА УЛИЦА
8	МОСКОВСКОЕ ШОССЕ УЛИЦА

В таблице А.2 приведены данные таблицы Abonent.

Таблица А.2 – Данные таблицы Abonent

ACCOUNTCD	STREETCD	HOUSENO	FLATNO	ФИО	PHONE
005488	3	4	1	АКСЕНОВ С.А.	556893
015527	3	1	65	КОНЮХОВ В.С.	761699
080047	8	39	36	ШУБИНА Т.П.	257842
080270	6	35	6	ТИМОШКИНА Н.Г.	321002
080613	8	35	11	ЛУКАШИНА Р.М.	254417
115705	3	1	82	МИЩЕНКО Е.В.	769975
126112	4	7	11	МАРКОВА В.П.	683301
136159	7	39	1	СВИРИНА З.А.	350003
136160	4	9	15	ШМАКОВ С.В.	982222
136169	4	7	13	ДЕНИСОВА Е.К.	680305
443069	4	51	55	СТАРОДУБЦЕВ Е.В.	683014
443690	7	5	1	ТУЛУПОВА М.И.	214833

В таблице А.3 приведены данные таблицы Services.

Таблица А.3 – Данные таблицы Services

GAZSERVICECD	GAZSERVICENM
1	Доставка газа
2	Заявочный ремонт ГО

В таблице А.4 приведены данные таблицы Executor.

Таблица А.4 – Данные таблицы Executor

EXECUTORCD	ФИО
1	СТАРОДУБЦЕВ Е.М.
2	БУЛГАКОВ Т.И.
3	ШУБИН В.Г.
4	ШЛЮКОВ М.К.
5	ШКОЛЬНИКОВ С.М.

В таблице А.5 приведены данные таблицы Disrepair.

Таблица А.5 – Данные таблицы Disrepair

FAILURECD	FAILURENM
1	Засорилась водогрейная колонка
2	Не горит АГВ
3	Течет из водогрейной колонки
4	Неисправна печная горелка
5	Неисправен газовый счетчик
6	Плохое поступление газа на горелку плиты
7	Туго поворачивается пробка крана плиты
8	При закрытии краника горелка плиты не гаснет
12	Неизвестна

В таблице А.6 приведены данные таблицы Request.

Таблица А.6 – Данные таблицы Request

REQUESTCD	ACCOUNTCD	EXECUTORCD	FAILURECD	INCOMINGDATE	EXECUTIONDATE	EXECUTED
1	5488	1	1	17.12.2001	20.12.2001	1
2	115705	3	1	07.08.2001	12.08.2001	1
3	15527	1	12	28.02.1998	08.03.1998	0
5	80270	4	1	31.12.2001	null	0
6	80613	1	6	16.06.2001	24.06.2001	1
7	80047	3	2	20.10.1998	24.10.1998	1
9	136169	2	1	06.11.2001	08.11.2001	1
10	136159	3	12	01.04.2001	03.04.2001	0
11	136160	1	6	12.01.1999	12.01.1999	1
12	443069	5	4	08.08.2001	10.08.2001	1
13	5488	5	8	04.09.2000	05.12.2000	1
14	5488	4	6	04.04.1999	13.04.1999	1
15	115705	4	5	20.09.2000	23.09.2000	1
16	115705	2	3	28.12.2001	null	0
17	115705	1	5	15.08.2001	06.09.2001	1
18	115705	2	3	28.12.1999	04.01.2000	1
19	80270	4	8	17.12.2001	27.12.2001	1
20	80047	3	2	11.10.2001	11.10.2001	1
21	443069	1	2	13.09.2001	14.09.2001	1
22	136160	1	7	18.05.2001	25.05.2001	1
23	136169	5	7	07.05.2001	08.05.2001	1

В таблице А.7 приведены данные таблицы NachislSumma.

Таблица А.7 – Данные таблицы NachislSumma

NACHISLFACTCD	ACCOUNTCD	GAZSERVICECD	NACHISLSUM	NACHISLMONTH	NACHISLYEAR
1	136160	2	56,00	1	1 999
2	5488	2	46,00	12	2 000
3	5488	2	56,00	4	1 999
4	115705	2	40,00	1	2 000
5	115705	2	250,00	9	2 001
6	136160	1	18,30	1	1 998
7	80047	2	80,00	10	1 998
8	80047	2	80,00	10	2 001
9	80270	2	46,00	12	2 001
10	80613	2	56,00	6	2 001
11	115705	2	250,00	9	2 000
12	115705	2	58,70	8	2 001
13	136160	2	20,00	5	2 001
15	136169	2	20,00	5	2 001
16	136169	2	58,70	11	2 001
17	443069	2	80,00	9	2 001
18	443069	2	38,50	8	2 001
19	5488	2	58,70	12	2 001
20	15527	1	28,32	7	1 998
21	80047	1	19,56	3	1 998
22	80613	1	10,60	9	1 998
23	443069	1	38,28	12	1 998
24	15527	1	38,32	4	1 999
25	115705	1	37,15	10	1 999
26	80613	1	12,60	8	2 000
27	136169	1	25,32	1	1 999
28	80270	1	57,10	2	1 998
29	136159	1	8,30	8	1 999
30	5488	1	62,13	4	2 000
31	115705	1	37,80	5	2 001
32	443690	1	17,80	6	1 998
33	80047	1	22,56	5	1 999
34	126112	1	15,30	8	2 000
35	80047	1	32,56	9	2 001
36	80613	1	12,60	4	1 998
37	115705	1	37,15	11	1 999
38	80270	1	58,10	12	2 000
39	136169	1	28,32	1	2 001
40	15527	1	18,32	2	1 998
41	443690	1	21,67	3	1 999
42	80613	1	22,86	4	2 000
43	80270	1	60,10	5	2 001
44	136169	1	28,32	2	1 998
45	80047	1	22,20	7	1 999
46	126112	1	25,30	8	2 001
47	443069	1	38,32	9	2 001
48	136159	1	8,30	10	1 998
49	115705	1	37,15	6	1 999
50	136160	1	18,30	12	2 000

В таблице А.8 приведены данные таблицы PaySumma.

Таблица А.8 – Данные таблицы PaySumma

PAYFACTCD	ACCOUNTCD	GAZSERVICECD	PAYSUM	PAYDATE	PAYMONTH	PAYYEAR
1	5488	2	58,70	08.01.2002	12	2 001
2	5488	2	46,00	06.01.2001	12	2 000
3	5488	2	56,00	06.05.1999	4	1 999
4	115705	2	40,00	10.02.2000	1	2 000
5	115705	2	250,00	03.10.2001	9	2 001
6	136160	2	20,00	13.06.2001	5	2 001
7	136160	2	56,00	12.02.1999	1	1 999
8	136169	2	20,00	22.06.2001	5	2 001
9	80047	2	80,00	26.11.1998	10	1 998
10	80047	2	80,00	21.11.2001	10	2 001
11	80270	2	46,00	03.01.2002	12	2 001
12	80613	2	56,00	19.07.2001	6	2 001
13	115705	2	250,00	06.10.2000	9	2 000
14	115705	2	58,70	04.09.2001	8	2 001
15	136169	2	58,70	01.12.2001	11	2 001
16	443069	2	80,00	03.10.2001	9	2 001
17	443069	2	38,50	13.09.2001	8	2 001
18	136160	1	18,30	05.02.1998	1	1 998
19	15527	1	28,32	03.08.1998	7	1 998
20	80047	1	19,56	02.04.1998	3	1 998
21	80613	1	10,60	03.10.1998	9	1 998
22	443069	1	38,28	04.02.1999	12	1 998
23	15527	1	38,32	07.05.1999	4	1 999
24	115705	1	37,15	04.11.1999	10	1 999
25	80613	1	12,60	20.09.2000	8	2 000
26	136169	1	25,32	03.02.1999	1	1 999
27	80270	1	57,10	05.03.1998	2	1 998
28	136159	1	8,30	10.09.1999	8	1 999
29	5488	1	62,13	03.05.2000	4	2 000
30	115705	1	37,80	12.07.2001	5	2 001
31	443690	1	17,80	10.07.1998	6	1 998
32	80047	1	22,56	25.06.1999	5	1 999
33	126112	1	15,30	08.09.2000	8	2 000
34	80047	1	32,56	18.10.2001	9	2 001
35	80613	1	12,60	22.05.1998	4	1 998
36	115705	1	37,15	23.12.1999	11	1 999
37	80270	1	58,10	07.01.2001	12	2 000
38	136169	1	28,32	08.02.2001	1	2 001
39	15527	1	18,32	18.03.1998	2	1 998
40	443690	1	21,67	10.04.1999	3	1 999
41	80613	1	22,86	04.05.2000	4	2 000
42	80270	1	60,10	07.06.2001	5	2 001
43	136169	1	28,32	05.03.1998	2	1 998
44	80047	1	22,20	10.08.1999	7	1 999
45	126112	1	25,30	10.09.2001	8	2 001
46	443069	1	38,32	09.10.2001	9	2 001
47	136159	1	8,30	14.11.1998	10	1 998
48	115705	1	37,15	10.08.1999	6	1 999
49	136160	1	18,30	07.01.2001	12	2 000

## Приложение Б

### Скрипт для создания учебной базы данных

```
SET SQL DIALECT 3;
CREATE DATABASE 'C:\SQLLAB.FDB' USER 'SYSDBA' PASSWORD
'masterkey'
PAGE_SIZE 4096 DEFAULT CHARACTER SET WIN1251;
/*****
/*          Domains          */
*****/
CREATE DOMAIN BOOLEAN AS SMALLINT CHECK (VALUE IN (0, 1));
CREATE DOMAIN MONEY AS NUMERIC(15,2);
CREATE DOMAIN TMONTH AS SMALLINT
CHECK (VALUE BETWEEN 1 AND 12);
CREATE DOMAIN PKFIELD AS INTEGER;
CREATE DOMAIN TYEAR AS SMALLINT
CHECK (VALUE BETWEEN 1990 AND 2100);
/*****
/*          Exceptions          */
*****/
CREATE EXCEPTION INS_RESTRICT 'Ограничение добавления записи в
дочернюю таблицу';
CREATE EXCEPTION DEL_RESTRICT 'Ограничение удаления записи из
родительской таблицы';
CREATE EXCEPTION UPD_RESTRICT 'Ограничение модификации записи в
родительской таблице';
/*****
/*          Tables          */
*****/
CREATE TABLE STREET (
    STREETCD PKFIELD NOT NULL PRIMARY KEY,
    STREETNM VARCHAR(30) );
CREATE TABLE SERVICES (
    GAZSERVICECD PKFIELD NOT NULL PRIMARY KEY,
    GAZSERVICENM VARCHAR(30) );
CREATE TABLE DISREPAIR (
    FAILURECD PKFIELD NOT NULL PRIMARY KEY,
    FAILURENM VARCHAR(50) );
CREATE TABLE EXECUTOR (
    EXECUTORCD PKFIELD NOT NULL PRIMARY KEY,
    FIO VARCHAR(20) );
CREATE TABLE ABONENT (
    ACCOUNTCD VARCHAR(6) NOT NULL PRIMARY KEY,
    STREETCD PKFIELD REFERENCES STREET
```



```

    ON DELETE SET NULL ON UPDATE CASCADE,
    HOUSENO SMALLINT,
    FLATNO SMALLINT,
    FIO VARCHAR(20),
    PHONE VARCHAR(15));
CREATE TABLE NACHISLSUMMA (
    NACHISLFACTCD PKFIELD NOT NULL PRIMARY KEY,
    ACCOUNTCD VARCHAR(6) NOT NULL REFERENCES ABONENT
    ON UPDATE CASCADE,
    GAZSERVICECD PKFIELD NOT NULL REFERENCES SERVICES
    ON UPDATE CASCADE,
    NACHISLSUM MONEY,
    NACHISLMONTH TMONTH,
    NACHISLYEAR TYEAR);
CREATE TABLE PAYSUMMA (
    PAYFACTCD PKFIELD NOT NULL PRIMARY KEY,
    ACCOUNTCD VARCHAR(6) NOT NULL REFERENCES ABONENT
    ON UPDATE CASCADE,
    GAZSERVICECD PKFIELD NOT NULL REFERENCES SERVICES
    ON UPDATE CASCADE,
    PAYSUM MONEY,
    PAYDATE DATE,
    PAYMONTH TMONTH,
    PAYYEAR TYEAR);
CREATE TABLE REQUEST (
    REQUESTCD PKFIELD NOT NULL PRIMARY KEY,
    ACCOUNTCD VARCHAR(6) REFERENCES ABONENT
    ON DELETE SET NULL ON UPDATE CASCADE,
    EXECUTORCD PKFIELD REFERENCES EXECUTOR
    ON DELETE SET NULL ON UPDATE CASCADE,
    FAILURECD PKFIELD REFERENCES DISREPAIR
    ON DELETE SET NULL ON UPDATE CASCADE,
    INCOMINGDATE DATE,
    EXECUTIONDATE DATE,
    EXECUTED BOOLEAN);
/*****
/*          Insert STREET          */
*****/
INSERT INTO STREET (STREETCD, STREETNM) VALUES (3, 'ВОЙКОВ
ПЕРЕУЛОК');
INSERT INTO STREET (STREETCD, STREETNM) VALUES (7, 'КУТУЗОВА
УЛИЦА');
INSERT INTO STREET (STREETCD, STREETNM) VALUES (6,
'МОСКОВСКАЯ УЛИЦА');

```

```

INSERT INTO STREET (STREETCD, STREETNM) VALUES (8,
'МОСКОВСКОЕ ШОССЕ УЛИЦА');
INSERT INTO STREET (STREETCD, STREETNM) VALUES (4, 'ТАТАРСКАЯ
УЛИЦА');
INSERT INTO STREET (STREETCD, STREETNM) VALUES (5, 'ТАГАРИНА
УЛИЦА');
INSERT INTO STREET (STREETCD, STREETNM) VALUES (1,
'ЦИОЛКОВСКОГО УЛИЦА');
INSERT INTO STREET (STREETCD, STREETNM) VALUES (2, 'НОВАЯ
УЛИЦА');
COMMIT WORK;
/*****
/*
Insert SERVICES
*/
/*****
INSERT INTO SERVICES (GAZSERVICECD, GAZSERVICENM) VALUES (1,
'Доставка газа');
INSERT INTO SERVICES (GAZSERVICECD, GAZSERVICENM) VALUES (2,
'Заявочный ремонт ГО');
COMMIT WORK;
/*****
/*
Insert DISREPAIR
*/
/*****
INSERT INTO DISREPAIR (FAILURECD, FAILURENM) VALUES (1,
'Засорилась водогрейная колонка');
INSERT INTO DISREPAIR (FAILURECD, FAILURENM) VALUES (2, 'Не горит
АГВ');
INSERT INTO DISREPAIR (FAILURECD, FAILURENM) VALUES (3, 'Течет из
водогрейной колонки');
INSERT INTO DISREPAIR (FAILURECD, FAILURENM) VALUES (4,
'Неисправна печная горелка');
INSERT INTO DISREPAIR (FAILURECD, FAILURENM) VALUES (5,
'Неисправен газовый счетчик');
INSERT INTO DISREPAIR (FAILURECD, FAILURENM) VALUES (6, 'Плохое
поступление газа на горелку плиты');
INSERT INTO DISREPAIR (FAILURECD, FAILURENM) VALUES (7, 'Туго
поворачивается пробка крана плиты');
INSERT INTO DISREPAIR (FAILURECD, FAILURENM) VALUES (8, 'При
закрытии краника горелка плиты не гаснет');
INSERT INTO DISREPAIR (FAILURECD, FAILURENM) VALUES (12,
'Неизвестна');
COMMIT WORK;
/*****
/*
Insert EXECUTOR
*/
/*****

```

```

INSERT INTO EXECUTOR (EXECUTORCD, FIO) VALUES (1,
'СТАРОДУБЦЕВ Е.М.');
```

```

INSERT INTO EXECUTOR (EXECUTORCD, FIO) VALUES (2, 'БУЛГАКОВ
Т.И.');
```

```

INSERT INTO EXECUTOR (EXECUTORCD, FIO) VALUES (3, 'ШУБИН В.Г.');
```

```

INSERT INTO EXECUTOR (EXECUTORCD, FIO) VALUES (4, 'ШЛЮКОВ
М.К.');
```

```

INSERT INTO EXECUTOR (EXECUTORCD, FIO) VALUES (5,
'ШКОЛЬНИКОВ С.М.');
```

```

COMMIT WORK;
/*****
/*          Insert ABONENT          */
*****/
INSERT INTO ABONENT (ACCOUNTCD, STREETCD, HOUSENO, FLATNO,
FIO, PHONE) VALUES ('005488', 3, 4, 1, 'АКСЕHOB C.A.', '556893');
```

```

INSERT INTO ABONENT (ACCOUNTCD, STREETCD, HOUSENO, FLATNO,
FIO, PHONE) VALUES ('115705', 3, 1, 82, 'МИЩЕHKO E.B.', '769975');
```

```

INSERT INTO ABONENT (ACCOUNTCD, STREETCD, HOUSENO, FLATNO,
FIO, PHONE) VALUES ('015527', 3, 1, 65, 'KOHIOXOB B.C.', '761699');
```

```

INSERT INTO ABONENT (ACCOUNTCD, STREETCD, HOUSENO, FLATNO,
FIO, PHONE) VALUES ('443690', 7, 5, 1, 'TYJYIOBA M.H.', '214833');
```

```

INSERT INTO ABONENT (ACCOUNTCD, STREETCD, HOUSENO, FLATNO,
FIO, PHONE) VALUES ('136159', 7, 39, 1, 'CBИPIHA 3.A.', '350003');
```

```

INSERT INTO ABONENT (ACCOUNTCD, STREETCD, HOUSENO, FLATNO,
FIO, PHONE) VALUES ('443069', 4, 51, 55, 'CTAPODYBЦEВ E.B.', '683014');
```

```

INSERT INTO ABONENT (ACCOUNTCD, STREETCD, HOUSENO, FLATNO,
FIO, PHONE) VALUES ('136160', 4, 9, 15, 'ШИAKOB C.B.', '982222');
```

```

INSERT INTO ABONENT (ACCOUNTCD, STREETCD, HOUSENO, FLATNO,
FIO, PHONE) VALUES ('126112', 4, 7, 11, 'MAPKOBA B.H.', '683301');
```

```

INSERT INTO ABONENT (ACCOUNTCD, STREETCD, HOUSENO, FLATNO,
FIO, PHONE) VALUES ('136169', 4, 7, 13, 'ДЕНИCOBA E.K.', '680305');
```

```

INSERT INTO ABONENT (ACCOUNTCD, STREETCD, HOUSENO, FLATNO,
FIO, PHONE) VALUES ('080613', 8, 35, 11, 'JYKACIIIHA P.M.', '254417');
```

```

INSERT INTO ABONENT (ACCOUNTCD, STREETCD, HOUSENO, FLATNO,
FIO, PHONE) VALUES ('080047', 8, 39, 36, 'ШУБИHA T.H.', '257842');
```

```

INSERT INTO ABONENT (ACCOUNTCD, STREETCD, HOUSENO, FLATNO,
FIO, PHONE) VALUES ('080270', 6, 35, 6, 'TИMOШKИHA H.H.', '321002');
```

```

COMMIT WORK;
/*****
/*          Insert NACHISLSUMMA          */
*****/
INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,
GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)
VALUES (19, '005488', 2, 58.7, 12, 2001);
```

INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (2, '005488', 2, 46, 12, 2000);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (3, '005488', 2, 56, 4, 1999);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (4, '115705', 2, 40, 1, 2000);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (5, '115705', 2, 250, 9, 2001);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (13, '136160', 2, 20, 5, 2001);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (1, '136160', 2, 56, 1, 1999);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (15, '136169', 2, 20, 5, 2001);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (7, '080047', 2, 80, 10, 1998);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (8, '080047', 2, 80, 10, 2001);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (9, '080270', 2, 46, 12, 2001);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (10, '080613', 2, 56, 6, 2001);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (11, '115705', 2, 250, 9, 2000);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (12, '115705', 2, 58.7, 8, 2001);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (16, '136169', 2, 58.7, 11, 2001);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (17, '443069', 2, 80, 9, 2001);

INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (18, '443069', 2, 38.5, 8, 2001);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (6, '136160', 1, 18.3, 1, 1998);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (20, '015527', 1, 28.32, 7, 1998);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (21, '080047', 1, 19.56, 3, 1998);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (22, '080613', 1, 10.6, 9, 1998);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (23, '443069', 1, 38.28, 12, 1998);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (24, '015527', 1, 38.32, 4, 1999);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (25, '115705', 1, 37.15, 10, 1999);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (26, '080613', 1, 12.6, 8, 2000);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (27, '136169', 1, 25.32, 1, 1999);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (28, '080270', 1, 57.1, 2, 1998);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (29, '136159', 1, 8.3, 8, 1999);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (30, '005488', 1, 62.13, 4, 2000);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (31, '115705', 1, 37.8, 5, 2001);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (32, '443690', 1, 17.8, 6, 1998);

INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (33, '080047', 1, 22.56, 5, 1999);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (34, '126112', 1, 15.3, 8, 2000);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (35, '080047', 1, 32.56, 9, 2001);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (36, '080613', 1, 12.6, 4, 1998);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (37, '115705', 1, 37.15, 11, 1999);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (38, '080270', 1, 58.1, 12, 2000);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (39, '136169', 1, 28.32, 1, 2001);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (40, '015527', 1, 18.32, 2, 1998);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (41, '443690', 1, 21.67, 3, 1999);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (42, '080613', 1, 22.86, 4, 2000);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (43, '080270', 1, 60.1, 5, 2001);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (44, '136169', 1, 28.32, 2, 1998);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (45, '080047', 1, 22.2, 7, 1999);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (46, '126112', 1, 25.3, 8, 2001);  
 INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,  
 GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)  
 VALUES (47, '443069', 1, 38.32, 9, 2001);

```

INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,
GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)
VALUES (48, '136159', 1, 8.3, 10, 1998);
INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,
GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)
VALUES (49, '115705', 1, 37.15, 6, 1999);
INSERT INTO NACHISLSUMMA (NACHISLFACTCD, ACCOUNTCD,
GAZSERVICECD, NACHISLSUM, NACHISLMONTH, NACHISLYEAR)
VALUES (50, '136160', 1, 18.3, 12, 2000);
COMMIT WORK;
/*****
/*          Insert PAYSUMMA          */
*****/
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (1, '005488', 2, 58.7,
'01/08/2002', 12, 2001);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (2, '005488', 2, 46,
'01/06/2001', 12, 2000);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (3, '005488', 2, 56,
'05/06/1999', 4, 1999);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (4, '115705', 2, 40,
'02/10/2000', 1, 2000);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (5, '115705', 2, 250,
'10/03/2001', 9, 2001);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (6, '136160', 2, 20,
'06/13/2001', 5, 2001);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (7, '136160', 2, 56,
'02/12/1999', 1, 1999);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (8, '136169', 2, 20,
'06/22/2001', 5, 2001);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (9, '080047', 2, 80,
'11/26/1998', 10, 1998);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (10, '080047', 2, 80,
'11/21/2001', 10, 2001);

```

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (11, '080270', 2, 46, '01/03/2002', 12, 2001);  
 INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (12, '080613', 2, 56, '07/19/2001', 6, 2001);  
 INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (13, '115705', 2, 250, '10/06/2000', 9, 2000);  
 INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (14, '115705', 2, 58.7, '09/04/2001', 8, 2001);  
 INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (15, '136169', 2, 58.7, '12/01/2001', 11, 2001);  
 INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (16, '443069', 2, 80, '10/03/2001', 9, 2001);  
 INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (17, '443069', 2, 38.5, '09/13/2001', 8, 2001);  
 INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (18, '136160', 1, 18.3, '02/05/1998', 1, 1998);  
 INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (19, '015527', 1, 28.32, '08/03/1998', 7, 1998);  
 INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (20, '080047', 1, 19.56, '04/02/1998', 3, 1998);  
 INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (21, '080613', 1, 10.6, '10/03/1998', 9, 1998);  
 INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (22, '443069', 1, 38.28, '02/04/1999', 12, 1998);  
 INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (23, '015527', 1, 38.32, '05/07/1999', 4, 1999);  
 INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (24, '115705', 1, 37.15, '11/04/1999', 10, 1999);  
 INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (25, '080613', 1, 12.6, '09/20/2000', 8, 2000);



INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (26, '136169', 1, 25.32, '02/03/1999', 1, 1999);

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (27, '080270', 1, 57.1, '03/05/1998', 2, 1998);

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (28, '136159', 1, 8.3, '09/10/1999', 8, 1999);

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (29, '005488', 1, 62.13, '05/03/2000', 4, 2000);

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (30, '115705', 1, 37.8, '07/12/2001', 5, 2001);

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (31, '443690', 1, 17.8, '07/10/1998', 6, 1998);

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (32, '080047', 1, 22.56, '06/25/1999', 5, 1999);

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (33, '126112', 1, 15.3, '09/08/2000', 8, 2000);

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (34, '080047', 1, 32.56, '10/18/2001', 9, 2001);

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (35, '080613', 1, 12.6, '05/22/1998', 4, 1998);

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (36, '115705', 1, 37.15, '12/23/1999', 11, 1999);

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (37, '080270', 1, 58.1, '01/07/2001', 12, 2000);

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (38, '136169', 1, 28.32, '02/08/2001', 1, 2001);

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (39, '015527', 1, 18.32, '03/18/1998', 2, 1998);

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD, PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (40, '443690', 1, 21.67, '04/10/1999', 3, 1999);

```

INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (41, '080613', 1,
22.86, '05/04/2000', 4, 2000);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (42, '080270', 1, 60.1,
'06/07/2001', 5, 2001);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (43, '136169', 1,
28.32, '03/05/1998', 2, 1998);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (44, '080047', 1, 22.2,
'08/10/1999', 7, 1999);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (45, '126112', 1, 25.3,
'09/10/2001', 8, 2001);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (46, '443069', 1,
38.32, '10/09/2001', 9, 2001);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (47, '136159', 1, 8.3,
'11/14/1998', 10, 1998);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (48, '115705', 1,
37.15, '08/10/1999', 6, 1999);
INSERT INTO PAYSUMMA (PAYFACTCD, ACCOUNTCD, GAZSERVICECD,
PAYSUM, PAYDATE, PAYMONTH, PAYYEAR) VALUES (49, '136160', 1, 18.3,
'01/07/2001', 12, 2000);
COMMIT WORK;
/*****
/*          Insert REQUEST          */
*****/
INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD,
FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (1,
'005488', 1, 1, '12/17/2001', '12/20/2001', 1);
INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD,
FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (2,
'115705', 3, 1, '08/07/2001', '08/12/2001', 1);
INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD,
FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (3,
'015527', 1, 12, '02/28/1998', '03/08/1998', 0);
INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD,
FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (5,
'080270', 4, 1, '12/31/2001', NULL, 0);

```

INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (6, '080613', 1, 6, '06/16/2001', '06/24/2001', 1);  
 INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (7, '080047', 3, 2, '10/20/1998', '10/24/1998', 1);  
 INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (9, '136169', 2, 1, '11/06/2001', '11/08/2001', 1);  
 INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (10, '136159', 3, 12, '04/01/2001', '04/03/2001', 0);  
 INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (11, '136160', 1, 6, '01/12/1999', '01/12/1999', 1);  
 INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (12, '443069', 5, 4, '08/08/2001', '08/10/2001', 1);  
 INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (13, '005488', 5, 8, '09/04/2000', '12/05/2000', 1);  
 INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (14, '005488', 4, 6, '04/04/1999', '04/13/1999', 1);  
 INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (15, '115705', 4, 5, '09/20/2000', '09/23/2000', 1);  
 INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (16, '115705', 2, 3, '12/28/2001', NULL, 0);  
 INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (17, '115705', 1, 5, '08/15/2001', '09/06/2001', 1);  
 INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (18, '115705', 2, 3, '12/28/1999', '01/04/2000', 1);  
 INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (19, '080270', 4, 8, '12/17/2001', '12/27/2001', 1);  
 INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (20, '080047', 3, 2, '10/11/2001', '10/11/2001', 1);  
 INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD, FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES (21, '443069', 1, 2, '09/13/2001', '09/14/2001', 1);

```

INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD,
FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES
(22, '136160', 1, 7, '05/18/2001', '05/25/2001', 1);
INSERT INTO REQUEST (REQUESTCD, ACCOUNTCD, EXECUTORCD,
FAILURECD, INCOMINGDATE, EXECUTIONDATE, EXECUTED) VALUES
(23, '136169', 5, 7, '05/07/2001', '05/08/2001', 1);
COMMIT WORK;
/*****
/*                      Triggers                      */
*****/
SET TERM ^ ;
CREATE TRIGGER TD_ABONENT FOR ABONENT
ACTIVE AFTER DELETE POSITION 0
AS
DECLARE VARIABLE NUMROWS INTEGER;
BEGIN
    SELECT COUNT(*)
    FROM NACHISLSUMMA
    WHERE
        NACHISLSUMMA.ACCOUNTCD = OLD.ACCOUNTCD INTO
NUMROWS;
    IF (NUMROWS > 0) THEN
    BEGIN
        EXCEPTION Del_Restrict;
    END
    SELECT COUNT(*)
    FROM PAYSUMMA
    WHERE
        PAYSUMMA.ACCOUNTCD = OLD.ACCOUNTCD INTO NUMROWS;
    IF (NUMROWS > 0) THEN
    BEGIN
        EXCEPTION Del_Restrict;
    END
END
^
CREATE TRIGGER TD_SERVICES FOR SERVICES
ACTIVE AFTER DELETE POSITION 0
AS
DECLARE VARIABLE NUMROWS INTEGER;
BEGIN
    SELECT COUNT(*)
    FROM NACHISLSUMMA
    WHERE
        NACHISLSUMMA.GAZSERVICECD = OLD.GAZSERVICECD INTO
NUMROWS;

```

```
IF (NUMROWS > 0) THEN
BEGIN
  EXCEPTION Del_Restrict;
END
SELECT COUNT(*)
FROM PAYSUMMA
WHERE
  PAYSUMMA.GAZSERVICECD = OLD.GAZSERVICECD INTO
NUMROWS;
IF (NUMROWS > 0) THEN
BEGIN
  EXCEPTION Del_Restrict;
END
END
^
SET TERM ; ^
```

## Список литературы

1. Дейт К.Дж. Введение в системы баз данных. – 8-е изд.: Пер.с англ. – К.; М.; СПб.: Издательский дом "Вильямс", 2005. – 1327 с.
2. Конноли Т., Бегг К., Страчан А. Базы данных: проектирование, реализация и сопровождение. Теория и практика. – 2-е изд.: Пер. с англ. – М.: Издательский дом "Вильямс", 2001. – 1120 с.
3. <http://www.abonentplus.ru>
4. Марков А.С., Лисовский К.Ю. Базы данных. Введение в теорию и методологию: учеб. – М.: Финансы и статистика, 2004. – 511 с.
5. Проектирование и реализация баз данных Microsoft SQL Server 2000. Учебный курс Microsoft /Пер. с англ. – 3-е изд. – М.: Издательско-торговый дом "Русская редакция"; СПб.: Питер, 2006. – 512 с.
6. Маклаков С.В. Создание информационных систем с AllFusion Modeling Suite. – М.: ДИАЛОГ-МИФИ, 2003 – 432 с.
7. Карпова Т.С. Базы данных: модели, разработка, реализация. – СПб.: Питер, 2001. – 303 с.
8. Чекалов А.П. Базы данных: от проектирования до разработки приложений. – СПб.: "БХВ-Петербург", 2003. – 384 с.
9. <http://www.intuit.ru>
10. Кириллов В.В., Громов Г.Ю.. Структуризированный язык запросов (SQL). - [http://www.citforum.ru/database/sql\\_kg/index.shtml](http://www.citforum.ru/database/sql_kg/index.shtml).
11. Дж. Грофф, П. Вайнберг. Энциклопедия SQL. – 3-е изд. – СПб.: Питер, 2003. – 816 с.
12. <http://www.ibase.ru>
13. <http://www.ibphoenix.com>
14. <http://www.ibexpert.com/rus/>
15. <http://www.sqlly.com>
16. Андон Ф., Резниченко В. Язык запросов SQL: учебный курс. – СПб.: Питер; Киев: Издательская группа ВНУ, 2006. – 416 с.
17. Туманов В. Е., Гайфуллин Б. Н., Сгибнев В. Я. Введение в SQL для баз данных в архитектуре клиент/сервер: учеб. пособие. – Издательство "Интерфейс Пресс". – 190 с.
18. Борри Х. Firebird: руководство разработчика баз данных: Пер. с англ. – СПб.: БХВ-Петербург, 2006. – 1104 с.
19. Скляр А.Я. Введение в InterBase – М.: Горячая линия-Телеком, 2002. – 517 с.
20. <http://www.firebirdsql.org>
21. Клайн К. при участии Клайна Д. и Ханта Бр. SQL: справочник. – 2-е изд. / Пер с англ. – М.: КУДИЦ-ОБРАЗ, 2006. – 832 с.
22. Моисеенко С.И. SQL. Задачи и решения. – СПб.: Питер, 2006. – 256 с.
23. Бьюли А. Изучаем SQL: Пер.с англ. – СПб: Символ-Плюс, 2007. – 312 с.

24. Джудит С. Боуман, Сандра Л. Эмерсон, Марси Дарновски. Практическое руководство по SQL. – 3-е изд.: Пер. с англ.: учеб. пособие – М.: Издательский дом "Вильямс", 2001. – 336 с.
25. Кауффман Д., Матсик Б., Спенсер К. и др. SQL. Программирование – М.: БИНОМ ЛЗ, 2002. – 744 с.
26. <http://www.interbase-world.com>
27. Ковязин А.Н., Востриков С.М. Мир InterBase: архитектура, администрирование и разработка приложений баз данных в InterBase/Firebird/Yafill. – М.: КУДИЦ-ОБРАЗ, 2005. – 432 с.
28. Бондарь А.Г. Interbase и Firebird. Практическое руководство для умных пользователей и начинающих разработчиков. – СПб.: БХВ-Петербург, 2007. – 592 с.
29. <http://www.sql.ru>
30. <http://www.sql-ex.ru>

М а р к и н Александр Васильевич

Построение запросов и программирование на SQL

Редактор Р.К. Мангутова

Корректор С.В. Макушина

Подписано в печать . Формат бумаги 60 x 84 1/16.

Бумага газетная. Печать трафаретная. Усл. печ. л. 19,5.

Уч.-изд. л. 19,5. Тираж 150 экз. Заказ

Рязанский государственный радиотехнический университет.

390005, Рязань, ул. Гагарина, 59/1.

Редакционно-издательский центр РГРТУ.