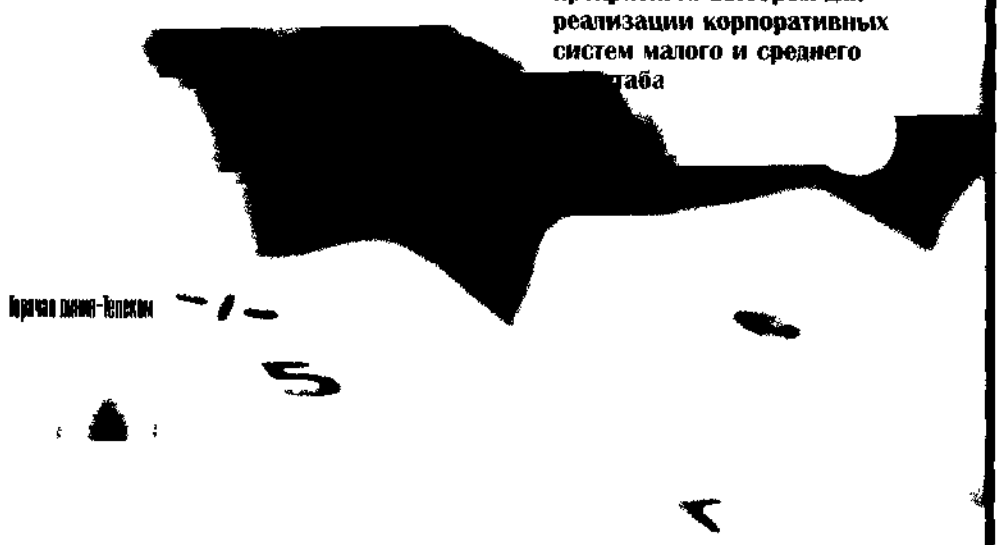
А.Я.СКЛЯР

ВВЕДЕНИЕ В InterBase

- **Высокая производительность и надежность сервера при минимальных требованиях к техническим средствам.**
- **Поддержка стандарта SQL-92, обеспечивающая переносимость приложений.**
- **Относительно низкая стоимость продукта.**
- **Простота установки и поддержки сервера. Удобный и не требующий специальной подготовки механизм администрирования базы данных.**

Все это делает InterBase прекрасным выбором для реализации корпоративных систем малого и среднего масштаба



ББК 32.973
С 43

Скляр А.Я.
С43 Введение в InterBase — М.: Горячая линия–Телеком, 2002. - 517 с: ил.
ISBN 5-93517-062-0.

Книга содержит справочные и методические материалы по популярной системе управления базами данных InterBase 5-6. Рассмотрена методика проектирования систем переработки информации на основе клиент - серверной технологии. Особое внимание уделено применению средств SQL при работе с данными, включая работу в многопользовательском режиме, поддержанию логической целостности данных, подробно освещен механизм транзакций, используемый в SQL-сервере InterBase. Изложена методика прикладного программирования на языке C++ для InterBase. Описаны Инструментальные средства для работы с InterBase. Справочный материал содержит полное описание языка SQL для InterBase, а также перечень диагностических сообщений, выдаваемых при работе сервера.

Книга рассчитана как на начинающих, так и на опытных разработчиков информационных систем, а также на студентов соответствующих специальностей.

ББК 32.973

Производственно-техническое издание

Скляр Александр Яковлевич
Введение в InterBase

Редактор Э.Н. Бадиков
Обложка художника В.Г. Ситникова

ЛР № 071825 от 16 марта 1999 г.
Подписано в печать 11.06.2002. Формат 60x88/16. Гарнитура Arial
Печать офсетная. Уч.-изд. л. 32,64. Тираж 3 000 экз. Изд. № 62. Заказ I* 2007

ISBN 5-93517-062-0

Введение

Базы данных в системах обработки информации

Автоматизация технологических и управленческих процессов, без которой немыслимо эффективное решение задач управления промышленным или торговым предприятием, банком, учебным заведением, государственной структурой, основывается на переработке больших объемов информации.

Эффективность автоматизированных информационных управляющих систем в значительной мере зависит от того, насколько обеспечивается скорость доступа к данным, их полнота, достоверность, непротиворечивость. И практически везде информационная система представляет собой интегрированную систему, ядро которой составляет база данных.

Персональные СУБД (Clipper, FoxPro, Clarion и др.) мало приспособлены для создания интегрированных систем, работающих с общей базой. В принципе эти СУБД вообще не поддерживают в строгом смысле понятие "база данных", работая на уровне индивидуальных таблиц-файлов и не обеспечивая контроля их логической целостности.

Более мощные системы, основанные на СУБД Btrieve, также не отвечают в полной мере требованиям масштабируемости, необходимой для корпоративной информационной системы. Достоинства Btrieve-систем, позволившие им унаследовать архитектуру и большую часть кода от своих предшественников, разработанных на Clipper и Clarion, что во многом объясняет их большую популярность, становятся тормозом при попытке построения информационных систем на более современных платформах и не обеспечивают переносимость решений.

Основным направлением в разработке автоматизированных информационных систем в настоящее время является ориентация на использование СУБД, базирующихся на SQL-серверах. В чем же состоят преимущества разработки информационных систем на их основе?

1. SQL-серверы прямо ориентированы на создание интегрированных, многопользовательских систем, имея в своем распоряжении развитые словари данных.
2. Средства разработки для этих СУБД оптимизированы в отношении коллективной разработки сложных систем в рамках единой стратегической линии.
3. Развитый механизм обработки транзакций позволяет обеспечить целостность данных при одновременной работе многих пользователей.
4. Использование единого языка доступа к данным (SQL) позволяет упростить переход от одной СУБД к другой.
5. Обеспечивается масштабируемость разрабатываемых систем.
6. Поддерживается возможность работы как в локальной, так и в глобальной сетях.

Рассматриваемая здесь СУБД InterBase в полной мере удовлетворяет всем перечисленным требованиям.

InterBase и область его применения

InterBase представляет собой полнофункциональный SQL-сервер. Сервер баз данных - это программный процесс, который выполняется на узле сети, где расположен главный компьютер и физически расположена сама база данных. Процесс сервера - единственный процесс на любом узле, который может исполнять прямые операции ввода-вывода для файлов базы данных.

Клиенты посылают запросы серверному процессу, чтобы выполнить различные действия в базе данных, включая:

- поиск в базе данных по заданным условиям;
- сравнение, сортировку и предоставление данных в табличном виде;
- изменение хранимых данных;
- добавление новых данных в базу;
- удаление данных из базы данных;
- создание новых базы данных и структур данных;
- выполнение программного кода на сервере;
- передачу сообщения другим клиентам, подключенным в данный момент к серверу.

Серверный процесс является полностью сетевым, он поддерживает запросы на подключение от других узлов сети и тот же самый протокол InterBase прикладной программы, что и клиентские процессы.

Несколько клиентов могут быть связаны с многопоточным процессом сервера одновременно.

Сервер регулирует доступ к отдельным записям данных в пределах базы данных и обеспечивает монопольный доступ к записям, когда клиенты выдают запросы на изменение данных в записях.

Отличительными качествами InterBase являются:

- Высокая производительность и надежность сервера при минимальных требованиях к техническим средствам.
- Поддержка стандарта SQL-92, обеспечивающая переносимость приложений.
- Относительно низкая стоимость продукта.
- Простота установки и поддержки сервера. Удобный и не требующий специальной подготовки механизм администрирования базой данных.

Все это делает InterBase прекрасным выбором для реализации корпоративных систем малого и среднего масштаба (с количеством пользователей в несколько десятков). При реализации очень крупных проектов (с сотнями или более пользователей) стоит, наверное, рассмотреть более мощные серверы - типа Oracle или Informix.

Системные требования InterBase

InterBase работает на различных платформах, включая Microsoft Windows NT 4.0, Windows 2000, Windows 95, Windows 98 и разные версии операционной системы UNIX.

Windows NT или Windows 95

Память: минимум 16 Мб (для сервера рекомендуется 64).

Процессор: 486DX2 66 МГц минимум; Pentium 100 МГц или больше рекомендуется для мультиклиентского сервера.

Компиляторы: Microsoft Visual C++ 4.2 и Borland **C++** 5.0, **C++** Builder, Delphi.

UNIX

Память: **минимум 16 Мб (для сервера** рекомендуется **64).**

Компиляторы: **Microsoft Visual C++ 4.2 и Borland C++ 5.0.**

HP-UX

Операционная система: HP-UX 10.20.

Должен быть установлен HP DCE/9000 - средство динамической поддержки (DCE-Core).

Память: минимум 32 Мб (для сервера рекомендуется 64).

Процессор: PA-RISC.

Компилятор C: HEWLETT-PACKARD C/HP-UX Версия 10.32.

Компилятор C++: C++ HEWLETT-PACKARD /HP-UX Версия 10.22.

Компилятор ФОРТРАНА: 10.20 выпуска HEWLETT-PACKARD Fortran/9000.

Аппаратная модель: HP/9000 Series 7xx или 8xx.

Solaris

Операционная система: Solaris 2.5.x или 2.6.x.

Память: минимум 32 Мб (для сервера рекомендуется 64).

Модель процессора: SPARC или UltraSPARC.

Компилятор C: SPARCWorks SC 4.2.

Компилятор C++: SPARCWorks SC3.0.1.

Компилятор ФОРТРАНА: SPARCWorks SC4.0.

Компилятор КОБОЛА: MicroFocus Cobol 4.0.

Компилятор Ады: SPARCWorks SC4.0 Ada compiler.

Основные возможности InterBase

InterBase на Windows 95 и Windows NT дает все выгоды от полной системы управления реляционной базой данных (RDBMS). Некоторые ключевые функции InterBase перечислены в следующей таблице.

Таблица. *Основные функции InterBase*

Функция	*Описание*
Поддержка сетевых протоколов	На всех платформах InterBase поддерживает TCP/IP; для Windows NT - каналы NetBEUI; для Netware - IPX/SPX
Соответствие минимальной конфигурации SQL-92	Стандартный ANSI SQL, доступный через утилиту интерактивного SQL (isql) и приложения Borland desktop

Функция	***Описание***
Доступ к базам данных	Одно приложение может обращаться к нескольким базам данных одновременно
Архитектура нескольких поколений	Сервер поддерживает (при необходимости) старые версии записей так, чтобы транзакции могли видеть непротиворечивое представление данных
Минимальный (оптимистический) уровень блокировки строк	Сервер блокирует только те записи, которые клиент модифицирует, вместо блокировки полной страницы базы данных
Оптимизация запросов	Сервер оптимизирует запросы автоматически. Можно также определить план запроса вручную
BLOB-данные и фильтры BLOB.	BLOB (большой двоичный объект) - данные, которые могут содержать неструктурированные данные типа графики или текста
Декларативная справочная целостность	Автоматическая поддержка логических связей между таблицами по внешним (FOREIGN) и первичным (PRIMARY) ключам
Хранимые процедуры	Программы, хранимые элементы в базе данных для расширения возможностей запросов на поиск и изменение данных
Триггеры	Программы, которые запускаются, когда в связанных с ними таблицах добавляются, модифицируются или удаляются данные
Индикация событий	Выдача сообщений приложению от базы данных. Дает возможность приложениям получить асинхронное уведомление об изменениях в базе данных
Обновляемые обзоры	Обзоры (виртуальные таблицы) могут отражать изменения данных сразу, как только они происходят

Функция	*Описание*
Определяемые пользователем функции (UDFs)	Программы (помещенные в специфицированную библиотеку на сервере), вызываемые из базы данных по запросам на SQL
Внешние объединения	Реляционная конструкция между двумя таблицами, которая допускает выполнение сложных операций
Явное управление транзакциями	Полное управление запуском, завершением или откатом транзакций, включая работу с поименованными транзакциями
Параллельный доступ приложений к данным	Один клиент, читающий таблицу, не блокирует доступ к таблице другим
Многомерные массивы	Столбец данных, размещаемый в индексированном списке элементов
Автоматическая двухфазная запись данных	Транзакционный контроль изменений в нескольких базах данных перед их окончательной записью или откатом (только для сервера InterBase)
InterBase API	Набор функций, которые дают возможность приложениям создавать операторы SQL/DSQL непосредственно для InterBase и сразу получать результаты
Gpre	Препроцессор для преобразования внедренных инструкций SQL/DSQL и переменных в формат, который может читаться компилятором базового языка
Server Manager (диспетчер сервера)	Windows утилита для резервного копирования базы данных, ее восстановления, обслуживания и защиты
Windows ISQL	Windows утилита для интерактивного определения данных и запросов
Isql	Утилита командной строки InterBase интерактивного SQL. Может использоваться вместо InterBase Windows ISQL

Функция	***Описание***
Утилита командной строки администратора базы данных (DBA)	Утилита командной строки InterBase административных средств базы данных; может использоваться вместо диспетчера сервера (Server Manager)
Заголовочные файлы (Header files)	Файлы, включаемые в начале прикладных программ и определяющие типы данных InterBase и сигнатуру функций обращения к InterBase
Примеры Файлов типа "make"	Файлы, которые демонстрируют, как создавать файлы для компиляции и компоновки InterBase приложений
Примеры программ	Программы на С, готовые к компиляции и компоновке, которые можно использовать для запросов к стандартному примеру базы данных сервера InterBase
Файл сообщений	Файл Interbase.msg, содержащий сообщения, представленные программам пользователя С, готовый к компиляции и компоновке, который может использоваться для запросов к стандартному примеру баз данных InterBase

Глава 1

Реляционные базы данных

1.1. Организация хранения данных

Для обеспечения эффективного хранения данных, а это означает быстрый поиск, обновление данных, защиту от ошибочных вводов, обеспечение конфиденциальности информации и многое другое, необходима соответствующая их организация. Для быстрого поиска необходимо упорядочение хранимых данных, поддержание связей между ними, контроль на непротиворечивость, обеспечение однократного ввода или изменения при многократном последующем использовании. Ключевую роль при этом играют методы поддержания логических связей между данными. По способам организации хранения связей выделяются такие модели данных, как иерархические, сетевые и реляционные.

Иерархическая модель

Первые иерархические и сетевые СУБД были созданы в начале 60-х годов, что было вызвано необходимостью управления миллионами записей (прежде всего связанных друг с другом иерархическим образом), например при информационной поддержке лунного проекта Аполлон. Среди реализуемых на практике СУБД этого типа наибольшее распространение получила система *IMS* (Information Management System компании IBM). Достаточно широко используются и другие иерархические системы.

Информация в иерархической модели данных представляется в виде совокупностей ориентированных деревьев. Формально *дерево* - это граф без циклов, в ориентированном дереве выделяется начальная вершина -

корень дерева. Движение по дереву осуществляется от корня. Графически дерево представляет собой множество точек - *вершин,* соединенных стрелками - *дугами,* причем в каждую вершину, кроме корня, входит в точности одна дуга, а исходит несколько (возможно, ни одной; тогда такую вершину называют концевой или *листом).* Таким образом, в иерархической модели реализуются связи *один ко многим.* В базах данных точкам соответствуют типы данных, а дугам - связи между данными разных типов.

Достоинства иерархической модели данных:

1. Относительная простота организации данных.
2. Высокая скорость доступа, как по поиску, так и по обновлению данных, имеющих по своей сути иерархическую структуру.

Ограничения иерархической модели:

1. Отсутствие явного разделения логических и физических характеристик модели.
2. Для представления неиерархических отношений данных требуются дополнительные манипуляции.
3. Непредвиденные запросы могут требовать реорганизации базы данных.

Сетевая модель

Сети - естественный способ представления отношений между объектами. Они широко применяются в математике, исследованиях операций, химии, физике, социологии и других областях знаний. Сети обычно могут быть представлены *ориентированным графом* общего вида. Графически ориентированный граф представляет собой множество точек - *вершин,* соединенных стрелками - *дугами.* В графе общего вида в каждую вершину может входить и выходить любое количество дуг. В контексте моделей данных узлы можно представлять как типы записей данных, а ребра представляют отношения один к одному или один ко многим. Структура графа делает возможным создание как простых представлений иерархических отношений (например, генеалогических данных), так и более сложных, в том числе рекурсивных.

Сетевая модель данных - это представление данных сетевыми структурами типов записей, связанных отношениями мощности один к одному или один ко многим. В конце 60-х годов конференция по языкам систем данных (Conference on Data Systems Languages, CODASYL) поручила группе DBTG (Database Task Group - группа для разработки стандартов систем управления базами данных) разработать стандарты систем управления базами данных. На DBTG оказывала сильное влияние архи-

тектура, использованная в одной из самых первых СУБД - Integrated Data Store (IDS), созданной ранее компанией General Electric. Это привело к тому, что была рекомендована сетевая модель.

Документы от 1971 года остаются основной формулировкой сетевой модели, на них ссылаются как на модель CODASYL DBTG. Она послужила основой для разработки сетевых систем управления базами данных нескольких производителей. IDS (Honeywell) и IDMS (Computer Associates) - две наиболее известные коммерческие реализации. В сетевой модели существует две основные структуры данных: типы записей и наборы.

- *Тип записей.* Совокупность логически связанных элементов данных.
- *Набор.* В модели DBTG отношение один ко многим между двумя типами записей.
- *Простая сеть.* Структура данных, в которой все бинарные отношения имеют мощность один ко многим.
- *Сложная сеть.* Структура данных, в которой одно или несколько бинарных отношений имеют мощность многие ко многим.
- *Тип записи связи.* Формальная запись, созданная для того, чтобы преобразовать сложную сеть в эквивалентную ей простую сеть.

В модели DBTG возможны только простые сети, в которых все отношения имеют мощность один к одному или один ко многим. Сложные сети, включающие одно или несколько отношений многие ко многим, не могут быть напрямую реализованы в модели DBTG. Следствием возможности создания искусственных формальных записей является необходимость дополнительного объема памяти и обработки, однако при этом модель данных имеет простую сетевую форму и удовлетворяет требованиям DBTG.

Достоинства сетевой модели данных:

4. Гибкая организации данных и, соответственно, возможность представления в базе самых разнообразных данных.
5. Возможность обеспечения исключительно высокой скорости поиска.

Ограничения сетевой модели:

6. Сложность процедур обновления данных, связанная с необходимостью перестроения системы связей.
7. Трудность поддержания логической целостности данных, высокий уровень риска при возникновении сбойных ситуаций.

Реляционная модель

В 1970-1971 годах Е.Ф. Кодд опубликовал две статьи, в которых ввел реляционную модель данных и реляционные языки обработки данных - реляционную алгебру и реляционное исчисление:

- *Реляционная алгебра* - процедурный язык обработки реляционных таблиц.
- *Реляционное исчисление* - непроцедурный язык создания запросов.

Все существующие к тому времени подходы к связыванию записей из разных файлов использовали физические указатели или адреса на диске. В своей работе Кодд продемонстрировал, что такие базы данных существенно ограничивают число типов манипуляций данными. Более того, они очень чувствительны к изменениям в физическом окружении. Когда в компьютерной системе устанавливался новый накопитель или изменялись адреса хранения данных, требовалось дополнительное преобразование файлов. Если к формату записи в файле добавлялись новые поля, то физические адреса всех записей файла изменялись. То есть такие базы данных не позволяли манипулировать данными так, как это позволяла бы логическая структура. Все эти проблемы преодолела **реляционная модель, основанная на логических отношениях данных.**

Существует два подхода к проектированию реляционной базы данных.

- *Первый подход* заключается в том, что на этапе концептуального проектирования создается не концептуальная модель данных, а непосредственно реляционная схема базы данных, состоящая из определений реляционных таблиц, подвергающихся нормализации.
- *Второй подход* основан на механическом преобразовании функциональной модели, созданной ранее, в нормализованную реляционную модель. Этот подход чаще всего используется при проектировании больших сложных схем баз данных, необходимых для корпоративных информационных систем.

1.2. Организация данных в реляционной модели

Рассматриваемая здесь СУБД InterBase относится к реляционным системам.

Классическая реляционная модель данных предполагает, что данные хранились в так называемых плоских таблицах. Фактически данные могут быть организованы и иначе, но пользователи и приложения, обращающиеся к данным, должны работать с данными так, как если бы они размещались в таких таблицах. В упрощенном виде плоская таблица - это

таблица, каждая ячейка которой может быть однозначно идентифицирована указанием строки и столбца таблицы. Кроме того, в одном столбце все ячейки должны содержать данные одного простого типа. Точное определение понятия "плоская таблица" дается в реляционной модели данных.

Реляционная модель основана на теории множеств и математической логике. Такой фундамент обеспечивает математическую строгость реляционной модели данных.

Отношения и кортежи

Подмножество $R = \{(a_1, a_2,\ldots,a_n) \mid a_i \in A_i\}$ декартового произведения множеств $A_1 \times A_2 \times \ldots \times A_n$ называется ***отношением степени n*** (***n-арным отношением***).

Элементы множества $r \in R$ $(r = (a_1, a_2,\ldots,a_n))$ называются кортежами.

Мощность множества кортежей, входящих в отношение R, называют ***мощностью отношения*** **R.**

Если множество R конечное, а при работе с базами данных это всегда так, то мощность - это просто число кортежей в отношении.

Понятие отношения фактически лежит в основе всей реляционной теории баз данных. Отношения являются математическим аналогом ***таблиц.*** В самом деле, множество i-ых элементов a_i кортежей можно трактовать как i-ый столбец таблицы, а сам кортеж - как ее строку. При этом природа отдельных элементов кортежей не имеет значения. Важно только то, что все они однотипны (принадлежат одному множеству A;).

Сам термин "реляционное представление данных", впервые введенный Коддом [43], происходит от термина *relation,* понимаемом именно в смысле этого определения.

Поскольку любое множество можно рассматривать как декартовое произведение степени 1, то любое подмножество, как и любое множество, можно считать отношением степени 1. Это не очень интересный пример, свидетельствующий лишь о том, что термины "отношение степени 1" и "подмножество" являются синонимами. Нетривиальность понятия отношения проявляется, когда степень отношения больше 1. Ключевыми здесь являются два момента: во-первых, все элементы отношения есть *однотипные* кортежи; во-вторых, за исключением крайнего случая, когда отношение есть само декартово произведение $A_1 \times A_2 \times \ldots \times A_n$, отношение включает в себя *не все возможные кортежи* из декартового произведения. Это значит, что отношение задает *критерий,* позволяющий определить, какие элементы декартова произведения $A_1 \times A_2 \times \ldots \times A_n$ удовлетворяют ему, а какие - нет. Этот критерий, по существу, и определяет *смысл (семантику)* отношения.

Однотипность кортежей позволяет считать их аналогами строк в простой таблице, т.е. в такой таблице, в которой все строки состоят из одинакового числа ячеек и в соответствующих ячейках содержатся одинаковые типы данных.

Например, отношение, состоящее из трех следующих кортежей {(1, "Иванов", 32), (2, "Петров", 41), (3, "Сидоров", 64)} можно считать таблицей, содержащей данные о сотрудниках и их возрасте. Такая таблица будет иметь три строки и три колонки, причем в каждой колонке содержатся данные одного типа. Заметим, что с практической точки зрения такая таблица, если это сведения о сотрудниках, плоха, поскольку данные в ней по мере прохождения времени придется обновлять, лучше указать год рождения; если же это данные о медицинском обследовании, то она оптимальна, иначе необходимо было бы давать и год рождения, и год обследования. С точки зрения анализа отношений, это, конечно, не имеет значения, а вот наличие первого столбца существенно. Если в нашу таблицу в дальнейшем будут добавляться строки, то нет никакой гарантии, что не появится Иванов "Второй". Для таблицы это несущественно, но все элементы множества, должны быть различимы, а значит, в нашем отношении не может быть двух одинаковых кортежей. Наличие же первого столбца с номером позволит нам легко отличить Иванова "Первого" от Иванова "Второго".

Операции с отношениями

Простое введение понятия отношения само по себе мало что дает. Необходимо определить, как с ним можно работать. Традиционно, вслед за Коддом, для отношений определяют восемь реляционных операторов, объединенных в две группы.

Теоретико-множественные операторы:

- Объединение
- Пересечение
- Вычитание
- Декартово произведение

Специальные реляционные операторы:

- Выборка
- Проекция
- Соединение
- Деление

Не все они являются независимыми, т.е. некоторые из этих операторов могут быть выражены через другие реляционные операторы.

Оператор объединения

Пусть R_1, $R_2 \subset A_1 \times A_2 \times \ldots \times A_n$, $a_i \in A_i$, тогда отношение $R = R_1 \cup R_2$ называется объединением R_1, R_2, т.е. $R = \{r_t = (a_{1t}, a_{2t}, \ldots, a_{nt}) \mid r_t \in R_1$ или $r_t \in R_2\}$.

Оператор пересечения

Пусть R_1, $R_2 \subset A_1 \times A_2 \times \ldots \times A_n$, $a_i \in A_i$, тогда отношение $R = R_1 \cap R_2$ называется пересечением R_1, R_2, т.е. $R = \{r_t = (a_{1t}, a_{2t}, \ldots, a_{nt}) \mid r_t \in R_1$ и $r_t \in R_2\}$.

Оператор вычитания

Пусть R_1, $R_2 \subset A_1 \times A_2 \times \ldots \times A_n$, $a_i \in A_i$, тогда отношение $R = R_1 \setminus R_2$ называется вычитанием R_1, R_2, т.е. $R = \{r_t = (a_{1t}, a_{2t}, \ldots, a_{nt}) \mid r_t \in R_1$ и $r_t \notin R_2\}$.

Оператор декартового произведения

Пусть $R_1 \subset A_1 \times A_2 \times \ldots \times A_n$, $R_2 \subset B_1 \times B_2 \times \ldots \times B_m$, $a_i \in A_i$, $b_i \in B_i$, тогда отношение $R = R_1 \times R_2$ называется декартовым произведением R_1, R_2, т.е. $R = \{r_t = (a_{1t}, a_{2t}, \ldots, a_{nt}, b_{1t}, b_{2t}, \ldots, b_{mt}) \mid (a_{1t}, a_{2t}, \ldots, a_{nt}) \in R_1$ и $(b_{1t}, b_{2t}, \ldots, b_{mt}) \in R_2\}$.

Оператор выборки

Пусть $R_1 \subset A_1 \times A_2 \times \ldots \times A_n$, $a_i \in A_i$, P – предикат на $A_1 \times A_2 \times \ldots \times A_n$, тогда отношение $R = (R_1, P) = \{r_t = (a_{1t}, a_{2t}, \ldots, a_{nt}) \mid r_t \in R_1$ и $P(r_t) =$ ***истина***$\}$ называется выборкой (селекцией) R_1 по P.

Оператор проекции

Пусть $R_1 \subset A_1 \times A_2 \times \ldots \times A_n$, $a_i \in A_i$, тогда отношение $R = R_1[k_1, k_2, \ldots, k_m] = \left\{ \begin{array}{l} r_i = (a_{k_i t}, a_{k_2 t}, \ldots, a_{k_m t}) \mid k_i \in (1+n) \text{ и для } \forall t \ \exists \text{кортеж} \\ r_s = (a_{1s}, a_{2s}, \ldots, a_{ns}) \in R_1, \text{ что } a_{k_i t} = a_{k_i s} \end{array} \right\}$ называется проекцией R_1 по $(k_1, k_2, \ldots, k_m)$.

Отметим, что если говорить в терминах таблиц, то проекция сводится к получению новой таблицы путем выбора из исходной некоторого множества столбцов и последующего удаления повторяющихся строк.

Оператор соединения

Оператор соединения определяется через операторы декартового произведения и выборки:

$R = ((S \times T), P)$

Для оператора естественного соединения добавляется оператор проекции, то есть

$R = ((S \times T), P) \{k_1, k_2, \ldots, k_m\}$

Удобнее его обозначать по другому. Пусть $S=\{s\}$, $T=\{t\}$ отношения на множествах столбцов (X,U) и (X,V) соответственно, то есть $s=(x_1, x_2,\ldots,x_f,u_1, u_2,\ldots,u_g)$, $t=(x_1, x_2,\ldots,x_f, v_1, v_2,\ldots,v_h)$, тогда $R = S \otimes_X T = \{ r_n = (x_{1n}, x_{2n},\ldots,x_{fn},u_{1n}, u_{2n},\ldots,u_{gn}, v_{1n}, v_{2n},\ldots,v_{hn}) \mid$ для $\forall n \exists s_k, t_m$, что для $i \in (1, f)$ $x_{in}=x_{ik}=x_{im}$ $(x_{in} \in r_n, x_{ik} \in s_k, x_{im} \in t_m)\}$.

Оператор деления

Пусть $R_1 \subset A_1 \times A_2 \times \ldots \times A_n \times B_1 \times B_2 \times \ldots \times B_m$, $R_2 \subset B_1 \times B_2 \times \ldots \times B_m$, тогда делением отношений R_1 и R_2 будем называть отношение $R \subset A_1 \times A_2 \times \ldots \times A_n$, $R = R_1/R_2$ такое, что $R \times R_2 \subset R_1$ и для $\forall R_3$, такого, что $R \subset R_3$ и $R_3 \neq R$, $R_3 \times R_2 \not\subset R_1$. Отношение R_1 выступает в роли ***делимого***, отношение R_2 в роли ***делителя***, отношение R в роли частного от деления. Деление отношений в определенном смысле аналогично делению с остатком для чисел.

Как уже отмечалось, перечисленные операторы являются, вообще говоря, зависимыми. Однако из них можно выделить группу независимых операторов. Например, группа: *объединение, вычитание, декартово произведение, выборка, проекция* или *объединение, пересечение.* Операторы такой группы называют примитивными. Состав таких операторов зависит от выбора базовой группы (либо первая, либо вторая).

Табличное представление данных; нормализация

Построение базы данных как системы взаимосвязанных таблиц является непростой задачей. Основное требование состоит в том, чтобы при построении базы данных избежать дублирования данных. При этом дело состоит не столько в росте объемов хранимой информации, сколько в трудности обеспечения их согласованности. Если данные о технических характеристиках оборудования будут храниться в нескольких таблицах, например таблицах, содержащих сведения об установленном оборудовании, таблицах наличия оборудования на складах, таблицах списанного оборудования, то изменения сведений об отдельном узле потребуется вносить во множество строк в нескольких таблицах. При этом трудно гарантировать полноту внесения изменений, что может привести к противоречиям при выборке и обработке данных, а это уже совсем недопустимо. Последнее означает, что все данные должны быть нормализованы с целью обеспечения отсутствия дублирования и, соответственно, однократности ввода или изменения при многократности использования данных. На практике, конечно, для ускорения поиска данных определенное

дублирование допускается. Важно лишь, чтобы оно было заранее известно и контролировалось. Обычно это относится к агрегированным данным или данным, полученным в результате специальной обработки.

Внешне таблицы, у которых изменен порядок расположения строк и столбцов, являются различными. С точки зрения обеспечения организации хранения данных это недопустимо. Поэтому в таблицах с каждым столбцом связывается его имя. Порядок строк в таблице не фиксируется (таблица отражает отношение, которое является множеством, а для элементов множества порядок не фиксируется). Таким образом, таблица определяется составом поименованных столбцов вне зависимости от порядка их записи, а также составом строк вне зависимости от порядка их следования.

Рассмотрим подробнее процедуры нормализации.

Первая нормальная форма

Таблица находится в ***Первой Нормальной Форме (1НФ),*** если она представляет собой отображение ***отношения***. Это означает, что:

- все данные в каждом из ее столбцов однотипны;
- данные каждого столбца элементарны (атомарны), то есть с точки зрения данного приложения не имеют внутренней структуры и должны обрабатываться как единое целое;
- все строки таблицы различны.

Перечисленные требования, по существу, тривиальны, тем не менее, они очень важны. Нарушение их практически делает неконтролируемой логическую целостность хранимых данных. Все более высокие формы нормализации таблиц базируются на **1НФ**.

Отношение представляет собой *множество* кортежей, поэтому отношения не могут содержать одинаковые кортежи. Это значит, что каждый кортеж должен обладать *свойством уникальности*. На самом деле свойством уникальности в пределах отношения могут обладать отдельные столбцы (атрибуты) кортежей или их группы. Такие уникальные атрибуты удобно использовать для идентификации кортежей.

Подмножество атрибутов ***K*** отношения ***R*** будем называть ***потенциальным ключом,*** если ***K*** обладает следующими свойствами:

1. ***Свойством уникальности*** - в отношении не может быть двух различных кортежей с одинаковым значением ***K***.
2. ***Свойством неизбыточности*** - никакое собственное подмножество в ***K*** не обладает свойством уникальности.

Любое отношение имеет, по крайней мере, один потенциальный ключ. Действительно, если никакой атрибут или группа атрибутов не яв-

ляются потенциальным ключом, то в силу уникальности кортежей все атрибуты вместе образуют потенциальный ключ.

Потенциальный ключ, состоящий из одного атрибута, называется ***простым.*** Потенциальный **ключ,** состоящий из **нескольких атрибутов,** называется ***составным.***

Отношение может иметь несколько потенциальных ключей. Традиционно один из потенциальных ключей объявляется ***первичным,*** а **остальные - *альтернативными.*** Различия между первичным и альтернативными ключами могут быть важны в конкретной реализации реляционной СУБД, но с точки зрения реляционной модели данных, нет оснований выделять таким образом один из потенциальных ключей.

Понятие потенциального ключа является *семантическим* понятием и отражает некоторый смысл (трактовку) понятий из конкретной предметной области. Последнее особенно важно в условиях, когда таблица будет в дальнейшем модифицироваться, поэтому самого факта, что на данный момент данная группа атрибутов (столбцов) годится в качестве потенциального ключа, недостаточно. Необходимо, чтобы можно было гарантировать эту уникальность и в дальнейшем.

Потенциальные ключи служат ***средством идентификации*** кортежей в отношении. Объекты предметной области, описываемые кортежами, должны быть различимы. При этом нужно помнить, что потенциальные ключи служат ***единственным*** средством адресации на уровне кортежей в отношении. Точно указать какой-нибудь кортеж можно, только зная значение его потенциального ключа.

Вторая Нормальная Форма

Отношение ***R*** находится во ***второй нормальной форме*** (***2НФ***) тогда и только тогда, когда отношение находится в ШФ и *нет неключевых атрибутов, зависящих от части сложного ключа.* (***Неключевой атрибут*** - это атрибут, не входящий в состав никакого потенциального ключа). Если некоторое множество атрибутов зависит от другого множества атрибутов, то первое множество будем называть зависимой частью функциональной зависимости, а второе - его детерминантом.

Отметим, что, если потенциальный ключ отношения является простым, то отношение автоматически находится в 2НФ.

Третья Нормальная Форма

Атрибуты называются ***взаимно независимыми,*** если ни один из **них** не является функционально зависимым от другого.

Отношение ***R*** находится в ***третьей нормальной форме (3НФ)*** тогда и только тогда, когда отношение находится в 2НФ и *все неключевые атрибуты взаимно независимы.*

Для того чтобы устранить зависимость неключевых атрибутов, нужно произвести декомпозицию отношения на несколько отношений. При этом *те неключевые атрибуты, которые являются зависимыми,* выносятся в отдельное отношение.

Нормальные формы высоких порядков

В большинстве случаев нормализации до уровня третьей нормальной формы (3НФ) вполне достаточно, чтобы разрабатывать вполне работоспособные базы данных. В то же время в некоторых случаях этого уровня оказывается недостаточно из-за трудностей с поддержанием логической целостности данных при обновлении информации. Связано это, прежде всего с тем, что в 3НФ могут сохраняться некоторые функциональные зависимости.

НФБК (Нормальная Форма Бойса-Кодда)

При приведении отношений с помощью алгоритма нормализации к отношениям в 3НФ неявно предполагалось, что все отношения содержат один потенциальный ключ. Это не всегда верно. Пусть отношение содержит два составных потенциальных ключа A+B и A+C и набор неключевых атрибутов D. C и B находятся в прямой зависимости друг от друга, тогда во всех кортежах, где встречается B, встречается и C. Налицо явная избыточность. В то же время, данное отношение находится во второй нормальной форме, поскольку оно не содержит *неключевых* атрибутов, зависящих от части сложного ключа. Более того, оно находится и в 3НФ, поскольку не содержит зависимых друг от друга *неключевых* атрибутов.

В то же время легко можно, разбив данное отношение на 2 (B, C) и (A, B, D) или (A, C, D), устранить избыточность. При таком действии устраняется зависимость между частями потенциального ключа.

Отношение *R* находится в ***нормальной форме Бойса-Кодда (НФБК)*** тогда и только тогда, когда *детерминанты всех функциональных зависимостей являются потенциальными ключами.*

4НФ и 5НФ (Четвертая и Пятая Нормальные Формы)

В ряде случаев зависимость между атрибутами может присутствовать, но при этом не быть однозначной. Например, если отношение (A, B, C) даже находится в нормальной форме Бойса-Кодда (НФБК), то это еще не означает, что его нельзя представить в виде эквивалентной пары отношений (A, B) и (A, C). В последнем случае говорят, что атрибуты (или группы атрибутов) B и C находятся в многозначной зависимости от A. Если такой зависимости не существует, а значит нельзя провести и эквивалентное разбиение отношения, то ***отношение находится в четвертой нормальной форме (4НФ).***

Эквивалентность группы отношений данному отношению означает, что соединение отношений этой группы совпадает с данным отношением.

Невозможность разбиения отношения на эквивалентную пару отношений опять же еще не означает отсутствие зависимостей внутри отношения. Пусть отношение образуют атрибуты (группы атрибутов) A, B, C, и отношение (A, B, C) находится в 4НФ. Однако может существовать разбиение отношения на группу из трех отношений (A, B), (A, C), (B,C), эквивалентных данному отношению. Если не существует разбиения данного отношения ни на какую группу отношений, эквивалентную данному, то ***отношение находится в пятой нормальной форме (5НФ).***

Рассмотрим на примерах приведение отношения к 4НФ и 5НФ. Пусть задана следующая таблица (отношение) со столбцами "товар", "поставщик", "потребитель". Соответствующие значения будем для краткости представлять кодами.

Таблица 1.1. *Отношение «товар-поставщик-потребитель»*

Товар	*Поставщик*	*Потребитель*
1	1	*2*
1	3	*2*
2	3	1
3	1	4
4	2	3
4	3	3

И пусть нам известно, что товары производятся не всеми поставщиками и используются не всеми потребителями. В этих условиях наше отношение можно разбить на пару отношений, эквивалентных данному.

Таблица 1.2. *Отношение «товар — поставщик»*

Товар	*Поставщик*
1	1
1	3
2	3
3	1
4	2
4	3

Таблица 1.3. *Отношение «товар - потребитель»*

Товар	*Потребитель*
1	2
2	1
3	4
4	3

Их соединение по столбцу "товар" дает исходную таблицу, которая изначально не находится в 4НФ и нормализуется данным разбиением.

Рассмотрим еще одну таблицу с аналогичными столбцами.

Таблица 1.4. *Отношение «товар-поставщик-потребитель»*

Товар	*Поставщик*	*Потребитель*
1	1	3
2	3	1
2	3	3
2	2	1
3	1	1
4	2	2

Данная таблица уже находится в 4НФ (не существует ее эквивалентного разбиения на пару таблиц вне зависимости от ее семантики). В то же время известно, что товары производятся не всеми поставщиками и используются не всеми потребителями и не все потребители имеют контакты с поставщиками. Соответствующие связи можно задать в виде трех отношений.

Таблица 1.5. *Отношение «товар-поставщик»*

Товар	*Поставщик*
1	1
2	3
2	2
3	1
4	2

Таблица 1.6. *Отношение «товар -потребитель»*

Товар	*Потребитель*
1	3
2	1
2	3
3	1
4	2

Таблица 1.7. *Отношение «поставщик- потребитель»*

Поставщик	*Потребитель*
1	3
3	1
3	3
2	1
1	1
2	2

Их последовательное соединение по столбцу "товар" и паре столбцов "поставщик", "потребитель" дает исходную таблицу, которая изначально не находится в 5НФ и нормализуется данным разбиением. (Отметим, что порядок соединения не влияет на результат).

Прежде чем двинуться дальше, сделаем несколько замечаний по содержанию 4НФ и 5НФ.

Во первых, нормализация к 4НФ и тем более к 5НФ возможна только в отношениях, не содержащих зависимых столбцов. Такие таблицы, хотя и встречаются не очень часто, но все же не являются совсем уж экзотикой. Их следует рассматривать как таблицы допустимых комбинаций составляющих их атрибутов. В реальных системах такие таблицы могут использоваться, например, для входного контроля данных.

Во вторых, нормализация данных к 4НФ, 5НФ, хотя далеко не всегда приводит к сокращению объемов хранимых данных, дает, тем не менее, то преимущество, что позволяет относительно безболезненно добавлять и удалять строки в соответствующих отношениях. При нарушении требований нормализации для этого пришлось бы делать групповые операции с дополнительными проверками, что при отсутствии явно выделенных смысловых зависимостей чревато ошибками.

Связи между данными, логическая целостность данных

Хранимые в базе данные таблицы логически связаны между собой. В частности, одним из результатов процедур нормализации данных является появление множества логически связанных таблиц. В общем случае можно говорить о наличии ***прямой связи между таблицами, если таблица А содержит первичный или уникальный ключ таблицы В.***

С другой стороны, сам по себе факт, что данные таблицы А детализируют данные таблицы В, никак не может помешать нам добавить в таблицу новые строки, для которых не будет соответствующих строк в таблице В. Например, если таблица В содержит перечень подразделений, а таблица А - список сотрудников, то нельзя гарантировать, что у нас не появятся сотрудники "никакого", то есть не описанного в таблице В, подразделения. В подобных случаях результаты работы программ могут оказаться неожиданными.

Такая же проблема возникает и при попытке добавить в таблицу строку с тем же первичным или уникальным ключом, что и у существующей.

Ряд полей (столбцов) могут принимать по своему смыслу не все значения, которые допустимы в соответствии с заданным для них типом, а должны принадлежать к определенному диапазону или перечню. Такие ограничения могут распространяться и на отдельное поле и на группу полей. Кроме того, между полями одной строки таблицы или данными разных таблиц могут существовать и более тонкие зависимости.

При традиционной файловой системе хранения данных функции контроля логической целостности данных возлагаются на прикладное программное обеспечение. Такой контроль осуществляется либо непосредственно, либо через соответствующие описания. В любом случае, нужно отметить, существует множество способов преднамеренно или случайно обойти этот контроль. Ошибки же в данных могут обойтись впоследствии очень дорого.

Отметим, что требования логической целостности относятся к самим данным, являются их свойством, а не свойством обрабатывающих их программ. Последнее означает, что значительно логичнее хранить описание требований логической целостности внутри базы данных и осуществлять их контроль средствами базы данных.

Для подобных целей в системах управления базами данных, в том числе и в InterBase, предусмотрены специальные средства.

Для недопущения дублирования значений первичных или альтернативных (уникальных) ключей блокируются любые попытки добавления в таблицу строк (записей), имеющих такие же значение ключей, как у существующих.

Для поддержания связей между таблицами в том смысле, в каком об этом говорилось выше, контролируются соответствующие значения ключевых полей, по которым осуществляется связь (внешние ключи). В этом случае запрещается добавление в дочернюю таблицу строк, для которых нет родительской. Запрещается также простое удаление тех строк (записей) родительской таблицы, для которых есть строки в дочерних таблицах. Здесь возможно несколько решений: простой запрет на удаление, удаление всех связанных строк в дочерних таблицах вместе с родительской, удаление строк родительской таблицы с одновременной пометкой связанных строк в дочерних таблицах как не имеющих родительской. Какой бы метод контроля при этом ни выбирался, гарантируется, что все хранящиеся в базе данные останутся согласованными между собой.

Помимо этого в ряде случаев необходимо контролировать на допустимость значения отдельных полей или их групп. Условия контроля могут быть достаточно сложными, в том числе требующими программной обработки.

Кроме того, дополнительный контроль, в том числе и связанный с анализом данных, хранящихся в разных таблицах, может осуществляться специальными программами-триггерами, которые будут включаться при попытке любого изменения данных. Для каждой такой попытки, а именно добавления (insert), модификации (update) и удаления (delete), можно задать свой триггер. Триггеры могут включаться как непосредственно перед соответствующим действием, так и после него. Общее количество триггеров не ограничено. Если триггеру "не понравятся" вводимые данные, он может выдать сообщение об ошибке и прервать соответствующую транзакцию.

Единственным ограничением на используемые методы контроля является то, что сам контроль требует определенных затрат времени и других ресурсов процессора. Поэтому в некоторых случаях массового ввода данных, про которые заранее известно, что они корректны, можно отключать средства контроля, что позволяет заметно сократить время загрузки данных. Правда, перед тем как принимать подобное решение, следует хорошо подумать, не приведет ли это к нежелательным последствиям.

Скорость доступа к данным

Скорость доступа к данным определяется объемом просматриваемой в процессе выборки информации. При отсутствии условий для выборки (фильтрации) и упорядочения просмотр осуществляется всегда в физическом порядке, обеспечивая максимальную скорость поиска. Когда задаются некоторые условия, то скорость поиска может быть существенно увеличена, если данные, участвующие в условиях выборки тем или иным способом упорядочены. В этом случае оказывается возможным сократить

объем просматриваемых данных, сразу исключив те, которые не удовлетворяют условиям выборки.

В InterBase упорядочение данных достигается заданием индексов по интересующему полю или группе полей. Индекс представляет собой список значений индексных полей, в котором каждому набору значений сопоставляется указатель на соответствующие строки таблиц. Сами индексы организуются, как правило, в виде бинарных деревьев или списков, что позволяет вести достаточно быстрый поиск по ним. При этом объем памяти, занимаемый индексом, относительно невелик.

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. Их можно перестроить, но это также требует затрат времени.

В этих условиях при проектировании базы данных необходимо находить разумный компромисс между требованиями по ускорению поиска данных и требованию по скорости их обновления. В частности, использование индексов для небольших по объему таблиц вообще не оправдано. Если имеется индекс по группе полей, то поиск по первому из полей групп может прямо использовать этот индекс, следовательно, нет смысла по нему делать отдельный индекс. Если поиск по каким-либо полям редок, то построение по ним индекса тоже не эффективно. К сожалению, более конкретные рекомендации по определению состава индексов едва ли возможны. Оптимальный выбор зависит и от структуры базы и от характера ее использования.

Глава 2

Основы языка SQL

2.1. Унификация доступа к данным

Нужно сразу отметить, что как бы эффективно не было организовано хранение данных, одного этого не достаточно для обеспечения решения проблемы реализации удобного и быстрого поиска и обновления информации. Огромное значение имеют алгоритмы процедур доступа к хранимым данным. Появление новых технических и программных решений влечет необходимость изменения алгоритмов поиска и обновления данных, но при этом необходимо обеспечить и возможность использования ранее разработанных программных продуктов. Кроме того, разработка прикладных систем не должна зависеть от выбора той или иной системы управления данными. Последнее особенно важно при изменении масштаба системы, что очень трудно предвидеть заранее, да и дополнительные затраты на ранних этапах попросту невыгодны. При этом очень хочется иметь возможность использовать полученные результаты и при новых разработках. Все это подталкивает к необходимости стандартизации, если не самих алгоритмов доступа к данным, что едва ли возможно без значительных потерь при переходе от одной системы управления базами данных (СУБД) к другой, то хотя бы описаний процедур доступа. В самом деле, если мы на каком-либо языке сумеем описать какие данные мы хотим получить или что мы хотим с ними сделать, а сам алгоритм будет строить компилятор соответствующей СУБД, то описание будет одним и тем же для всех СУБД, понимающих этот язык.

Именно эту проблему и решает язык SQL (Structured Query Language - язык структурированных запросов). Можно много спорить о его достоинствах и недостатках. Во многом он эклектичен и неудобен. Сходные команды имеют различный синтаксис, нет многих привычных по другим языкам функций, многие решения являются результатом не тщательного

анализа, а обычного компромисса между разными группами разработчиков СУБД. Словом, недостатков множество. Достоинство только одно, но оно стоит всех недостатков. На нем действительно можно описать требования к поиску и выборке данных, к их модификации, к логической структуре данных, методам поддержания логической целостности данных и их контролю. И, что самое главное, практически все фирмы разрабатывающие СУБД, приняли его как стандарт. Стандарт, конечно, развивается, но следующие версии языка всегда включают более ранние, как свою часть, что и дает необходимую совместимость. Благодаря этому разработчик может не заботиться о том, в среде какой СУБД будет работать его задача, если он, конечно, не использует слишком активно особенности конкретного диалекта SQL.

Сам язык SQL был разработан в 1970 году в компании IBM. В настоящее время он стал стандартным языком, используемым для связи с большинством систем управления базами данных, в том числе таких, как Oracle, INGRES, Informix, Sybase, SQLbase, Microsoft SQL Server, DB2, продуктами SQL/DC, Paradox, Access, Approach и многими другими.

Говоря о языке SQL нужно помнить о его главном назначении.

Во-первых, это описание запросов на поиск и изменение данных в существующей базе. Эту часть будем называть языком управления доступом к данным или языком манипулирования данными *(Data Manipulation Language - DML)*.

Во-вторых, это средства описания хранимых данных, их структуры, правил доступа к ним. Эту часть будем называть языком определения данных *(Data Definition Language - DDL)*.

В-третьих, это средства управления порядком доступа к данным. Эти средства образует группа операторов управления данными *(Data Definition Statements)*. Часто их относят к языку определения данных (DDL). Основные операторы данной группы - *GRANT* и *REVOKE*.

2.2. Язык управления доступом кданным

С точки зрения человека, пользующегося тем или иным хранилищем данных, существуют всего четыре действия над данными: поиск и выборка запрошенных данных, ввод новых данных, обновление существующих данных и, наконец, удаление данных, ставших ненужными. В соответствии с этим в SQL для решения этих задач и предусмотрены четыре команды:

Select - выборка данных, удовлетворяющих заданным условиям;

Insert - ввод новых данных;

Update - обновление существующих данных;

Delete - удаление данных.

Каждая из этих команд имеет множество вариантов, которые заслуживают отдельного рассмотрения.

2.3. Язык определения данных

Для того чтобы работать с данными, необходимо их сначала описать, то есть указать их структуру, формы представления, связи, методы их контроля и многое другое. О типах данных мы еще поговорим в дальнейшем, а пока отметим только, что действий с описаниями также весьма немного. Поскольку функция поиска здесь не имеет самостоятельного значения, то таких действий, по аналогии со сказанным о хранилище данных, остается всего три. Это ввод новых описаний, модификация существующих и удаление ненужных. Каждому из этих действий соответствует своя команда:

Create - ввод новых описаний;

Alter - модификация существующих описаний;

Drop - удаление ненужных описаний.

Каждая из этих команд имеет множество вариантов, связанных как с вариантами описаний, так и с тем фактом, что в описании нуждается множество различных информационных объектов.

2.4. Язык управления порядком доступа к данным

Поддержание целостности данных предполагает их защиту от случайного или намеренного искажения. Кроме того, доступ к отдельным данным должен быть ограничен и по соображениям их конфиденциальности. Все это требует ограничения прав доступа к данным различных групп пользователей. Последнее особенно важно в крупных системах с большим количеством пользователей. Предоставление соответствующих прав и их изъятие реализуется следующими командами.

Grant - предоставляет права доступа к специфицированным объектам данных со стороны указанных пользователей или других объектов базы.

Revoke - ликвидирует права доступа к специфицированным объектам данных со стороны указанных пользователей или других объектов базы.

2.5. Данные и метаданные

И еще одно замечание. Описания хранимых данных сами по себе также являются данными. А раз так, то нет никаких причин хранить их каким-либо особенным способом, отличным от основной информации. СУБД должна лишь априорно знать, где и в каком виде хранятся описания данных (или метаданные). Сами же процедуры работы с ними могут и ничем не отличаться от работы с обычными данными. Таким образом, отличительной особенностью всех современных СУБД, использующих SQL, является то, что и данные, и их описания хранятся в общей базе. Наличие же языка высокого уровня (SQL) позволяет в процессе работы интерпретировать написанные на нем запросы, освобождая пользователя от необходимости знать механизмы реальной обработки данных по их поиску, выборке и модификации.

2.6. Уровни реализации языка SQL

По самому своему существу SQL предназначен только для описания взаимодействия с базой данных. SQL не содержит средств, необходимых для разработки законченных программ и может использоваться только как их часть. В зависимости от назначения SQL может использовать один из трех уровней реализации.

1. **Интерактивный или автономный SQL (ISQL)** дает возможность пользователям непосредственно извлекать информацию из базы данных или записывать ее в базу данных. Эта уровень доступен посредством соответствующих утилит или специальных стандартных средств для работы с базами данных. В InterBase это реализуется такими утилитами, как Isql.exe и Wisql32.exe. С их помощью информация может быть загружена в базу данных или извлечена из нее и выведена на экран, печать или в файл.
2. **Статический SQL (SQL)** - фиксированный (исполнимый), то есть заранее подготовленный и записанный, а не формируемый в процессе выполнения программы SQL код, обычно используемый в приложениях. Существует две версии статического SQL. *Встроенный или внедренный SQL* - это код SQL, включенный в исходный текст программы, написанной на том или ином алгоритмическом языке. Используемый язык при этом называют базовым. InterBase поддерживает внедренный SQL. Другое использование статического SQL - *модульный язык*. В этом случае модули SQL присоединяются к приложению на этапе компоновки.

3. **Динамический SQL (DSQL)** - код SQL, генерируемый приложением во время его выполнения. Динамический SQL заменяет статический в тех случаях, когда необходимый код не может быть определен во время написания приложения, так как он сам зависит от действий пользователя во время выполнения приложения.

В InterBase используются все 3 уровня реализации SQL. Необходимо помнить, что на разных уровнях доступны не все команды языка. Кроме того, синтаксис команд разных уровней может отличаться. При описании синтаксиса соответствующих команд оговаривается уровень реализации, на который ориентирован этот синтаксис. В большинстве случаев, правда, этот синтаксис является единым.

2.7. Описание синтаксиса языка SQL

Для полного и однозначного описания синтаксических конструкций языка SQL необходимо определить сначала структурные элементы описания синтаксиса. Для этого нужен свой язык, такой чтобы можно было отличить собственно конструкции SQL от средств их описания. Примем следующие соглашения:

1. Ключевые слова и конструкции языка будем выделять **жирным шрифтом.**
2. Элементы описания будем выделять *курсивом.*
3. Если элемент или группа элементов являются необязательными и могут отсутствовать, то их будем помещать в *квадратные скобки [ ], выделенные курсивом.* Если квадратные скобки являются элементом SQL, то они будут заданы прямым шрифтом [].
4. Группа элементов, из которых только 1 должен присутствовать в конечной конструкции будем помещать в *фигурные скобки { },* а сами элементы разделять *вертикальной чертой* |.
5. Если какое-либо значение предполагается по умолчанию, то будем выделять его подчеркиванием.
6. Списки, состоящие из группы однородных элементов, будем обозначать в виде *LIST_*ElemName, где *LIST_* выступает в роли префикса, а содержание самого элемента будет описываться отдельно.
7. Описание элемента конструкции будем начинать с ElemName*::=*, после чего будем давать само описание.

8. Если элемент конструкции может быть составным, то будем его помещать в угловые скобки < >, иначе указывать непосредственно. Все элементы синтаксиса будем писать латиницей, учитывая общие требования к идентификаторам, используемым в SQL.

9. В соответствии с изложенным конструкция *LIST*_ElemName будет описываться как

*LIST*_ElemName::= ElemName *[,LIST*_ElemName*]*

Глава 3

Управление доступом в InterBase на основе SQL

3.1. Выборка данных. Команда SELECT

Назначение и основные возможности команды

Команда SELECT предназначена для выборки данных из базы. С ее помощью можно получить данные, удовлетворяющие заданным условием из одного или нескольких связанных объектов базы. Такими объектами являются, прежде всего, таблицы базы данных, но могут быть и обзоры, и хранимые процедуры, причем в любых сочетаниях. Выбранные данные могут быть агрегированы, отсортированы, с ними можно произвести ряд предварительных вычислений.

Базовый синтаксис команды SELECT

Дадим теперь полный синтаксис команды SELECT.

```
SELECT [TRANSACTION Transaction]
[{DISTINCT | ALL}] LIST_<val>
[INTO LIST_Vars]
 FROM LIST_<TableRef>
[WHERE <SearchCondition>]
[GROUP BY LIST_<Column>]
[HAVING <SearchCondition>]
```

```
[UNION <SelectExpr> [ALL] ]
[PLAN <PlanExpr>]
[ORDER BY LIST_<Orders>]
[FOR UPDATE [OF LIST_Col]];

<Val> ::= {
Col [<Array_dim>] | :Variable
| <Constant> | <Expr> | <Function>
| udf ([LIST_<Val>])
| NULL | USER | RDB$DB_KEY | ?
} [COLLATE Collation] [ [AS] Alias]
```

Col - имя столбца,

<Array_dim> ::= [LIST_Dim]

Dim ::= [x:]y

y - задает размерность массива.

x - задает нижнюю границу массива (если задано x:y, то индекс в масс иве меняется от x до y)

<constant> ::= число I ' строка' | Charsetname ' строка'

Charsetname - имя используемого набора символов, например WIN1251, WIN1252 и т.д.

<expr> ::= Любое корректное SQL выражение, дающее в результате единственное значение.

```
<function> ::= {
COUNT (* | [ALL] <val> | DISTINCT <val>)
| SUM ([ALL] <val> | DISTINCT <val>)
| AVG ([ALL] <val> | DISTINCT <val>)
| MAX ([ALL] <val> | DISTINCT <val>)
| MIN ([ALL] <val> | DISTINCT <val>)
| CAST ( <val> AS <datatype>)
| UPPER ( <val>)
| GEN_ID (generator, <val>)
}
```

В версии InterBase 6 добавлена также такая функция как:

```
EXTRACT(part FROM <DTExpr>)
```

Управление доступом в InterBase на основе SQL 35

part ::= **DAY** | HOUR | MINUTE | MONTH | SECOND | WEEKDAY | YEAR | YEARDAY }

<DTExpr> ::= Любое корректное SQL выражение, дающее в результате единственное значение типа дата - время.

Vars ::= :Var

<tableref> ::= <joined_table> | table | view | procedure [([LIST_<Val>])] [alias]

<joined_table> ::= <tableref> <join_type> JOIN <tableref> **ON** <search_condition> | (<joined_table>)

<join-type> ::= { [**INNER**] | {**LEFT** | RIGHT | **FULL**} [**OUTER**] } **JOIN**

<search_condition> ::= {<val> **<operator>** {<val> | (<select_one>)}

| <val> [**NOT**] **BETWEEN** <val> **AND** <val>

| <val> [**NOT**] **LIKE** <val> [**ESCAPE** <val>]

| <val> [**NOT**] **IN** ([LIST_<Val>] | <select_list>)

| <val> **IS [NOT] NULL**

| <val> {[**NOT**] { | **<** | **>**} | **>=** | **<=**}

{**ALL** | **SOME** | **ANY**} (<select_list>)

| **EXISTS** (<select_expr>)

| **SINGULAR** (<select_expr>)

| <val> [**NOT**] **CONTAINING** <val>

| <val> [**NOT**] **STARTING** [**WITH**] <val>

| (<search_condition>)

| **NOT** <search_condition>

| <search_condition> **OR <search_condition>**

| <search_condition> **AND** <search_condition>}

<operator> ::= { | < | > | <= | >= | !< | !> | <> | **!=**}

<select_one> ::= SELECT с одним столбцом, возвращающий одну строку,

<select_list> ::= SELECT с одним столбцом, возвращающий несколько (возможно 0) строк,

36 Глава 3

```
<select_expr>  ::=  SELECT  с  несколькими  столбцами,
возвращающий несколько (возможно 0) строк.
<plan_expr> ::=
[JOIN | MERGE] (LIST_<plans>)
<plans> ::= <plan_item> | <plan_expr>
<plan_item> ::= {table | alias}
NATURAL | INDEX (LIST_index) | ORDER index

<Orders> ::=
{col | int} [COLLATE collation] [ASC [ENDING] |
DESC [ENDING] ]
```

Отметим, что конструкции INTO и FOR UPDATE доступны только в процедурах и триггерах базы данных, либо во внедренном SQL в приложениях.

Конструкция команды SELECT достаточно сложная, поэтому разбор ее возможностей проведем поэтапно и приведем соответствующие примеры.

Выборка таблицы целиком

```
SELECT * FROM <tableref>;
```

* после SELECT означает выбор всего набора полей таблицы, имя которой задано в tableref.

Пример 3.1[1]

```
SELECT * FROM TREADER;
```

[1] В примерах используется главным образом база TESTBASE.GDB. Ее структура и содержание приведены в приложении. Случаи, когда для примеров используются другие данные, будут оговариваться отдельно.

Таблица 3.1. *Список читателей (включая служебные данные)*

UNIKEY	*RDNUMB*	*RDNAME*
36	1267-89	Арцибашев С.
37	1369-99	Светлова В.
38	1456-00	Стародуб Е.
39	1273-92	Гребенкина Н.
40	1400-00	Пащенко О.
83	1401-99	Грамотный Н.Е.

Данный пример для большинства применений явно плох. Выдается ряд данных сугубо служебного назначения, в лучшем случае бесполезных для конечного пользователя. Поэтому лучше явно указать именно те поля, которые его интересуют.

Выборка заданного списка полей таблицы

```
SELECT LIST_<val> FROM <tableref>;
```

список val1, val2, ... задает перечень имен полей таблицы, подлежащих выборке.

Пример 3.2

```
SELECT RDNUMB ,RDNAME FROM TREADER;
```

Таблица 3.2. *Список читателей (выборочные данные)*

RDNUMB	*RDNAME*
1267-89	Арцибашев С.
1369-99	Светлова В.
1456-00	Стародуб Е.
1273-92	Гребенкина Н.
1400-00	Пащенко О.
1401-99	Грамотный Н.Е.

А теперь отсортируем их по фамилиям, указав имя поля, по которому ведется сортировка,

```
SELECT RDNUMB ,RDNAME FROM TREADER ORDER BY RDNAME;
```

или просто его порядковый номер в списке полей выборки (иногда это единственно возможное решение).

Пример 3.3

```
SELECT RDNUMB ,RDNAME FROM TREADER ORDER BY 2;
```

Таблица 3.3. *Список читателей (выборочные отсортированные данные)*

RDNUMB	*RDNAME*
1267-89	Арцибашев С.
1401-99	Грамотный Н.Е.
1273-92	Гребенкина Н.
1400-00	Пащенко О.
1369-99	Светлова В.
1456-00	Стародуб Е.

Пока мы брали из таблицы все ее строки, но это нужно далеко не всегда. Выберем теперь только новых читателей (Номер билета >1300).

Пример 3.4

```
SELECT RDNUMB , RDNAME FROM TREADER where RDNUMB>='1400'
ORDER BY 2;
```

Таблица 3.4. *Выборка из списка читателей (выборочные отсортированные данные)*

RDNUMB	*RDNAME*
1401-99	Грамотный Н.Е.
1400-00	Пащенко О.
1456-00	Стародуб Е.

Согласно требованиям нормализации мы разбили данные на множество таблиц, но это разбиение никак не связано с тем, какие данные мы хотим получить. Последнее означает, что необходимо уметь выбирать данные одновременно из нескольких таблиц. Например, так

Пример 3.5

```
SELECT * FROM TREADER, TBOOK;
```

Таблица 3.5. *Выборка списка читателей и книг*

unikey	*rdnumb*	*rdname*	*uni-key1*	*mother-key*	*booknm*	*Re-ferat*
36	1267-89	Арцибашев С.	2	0	Программирование	
37	1369-99	Светлова В.	2	0	Программирование	
38	1456-00	Стародуб Е.	2	0	Программирование	
39	1273-92	Гребенкина Н.	2	0	Программирование	
40	1400-00	Пащенко О.	2	0	Программирование	
83	1401-99	Грамотный Н.Е.	2	0	Программирование	
36	1267-89	Арцибашев С.	3	0	Учебники	
37	1369-99	Светлова В.	3	0	Учебники	
...	...	...	...	...	...	

Информационная ценность такого выбора (прямое декартово произведение) явно невелика, выбирать следует только связанные между собой данные, например, связав читателей с взятыми ими книгами, явно указывая поля, которые нам интересны:

```
SELECT TREADER.rdnumb, TREADER.rdname, TBOOK.booknm
 FROM TREADER, TBOOK, TBOOK_READER
 where tbook_reader.reader=treader.unikey and
                  tbook_reader.bookkey=tbook.unikey;
```

Поскольку имена полей в разных таблицах могут повторяться, то их нужно уточнять, предваряя именем таблицы или заменяющим его алиасом, который конечно нужно объявить, например так

Пример 3.6

```
SELECT a.rdnumb, a.rdname, b.booknm
 FROM TREADER a, TBOOK b, TBOOK_READER ab
 where ab.reader=a.unikey and ab.bookkey=b.unikey;
```

Таблица 3.6. *Выборка списка читателей и взятых ими книг*

RDNUMB	*RDNAME*	*BOOKNM*
1267-89	Арцибашев С.	Тайна
1267-89	Арцибашев С.	Математические вопросы динамики вязкой несжимаемой жидкости
1267-89	Арцибашев С.	Тесты. Сборник 11 класс. Варианты и ответы государственного тестирования. Пособие для подготовки к тестированию
1369-99	Светлова В.	Тайна
1456-00	Стародуб Е.	Язык С++
1273-92	Гребенкина Н.	The history of England. Absolute Monarchy
1400-00	Пащенко О.	Введение в технологию АТМ
1400-00	Пащенко О.	Word 6 for Windows

Книги, правда, получились без авторов. Попробуем теперь вытащить фамилии авторов.

Пример 3.7

```
SELECT a.rdnumb, a.rdname, b.booknm, c.AUNAME
 FROM TREADER a, TBOOK b, TBOOK_READER ab,
      TAUTHOR c, TBOOK_AUTHOR bc
 where ab.reader=a.unikey and ab.bookkey=b.unikey
  and b.unikey= bc.bookkey and bc.author=c.author;
```

Таблица 3.7. *Выборка списка читателей и взятых ими книг с указанием авторов*

RDNUMB	*RDNAME*	*BOOKNM*	*AUNAME*
1400-00	Пащенко О.	Word 6 for Windows	Фаненштих Клаус
1400-00	Пащенко О.	Word 6 for Windows	Хаселир Райнер Г.
1456-00	Стародуб Е.	Язык С++	Подбельский Вадим Валериевич

RDNUMB	*RDNAME*	*BOOKNM*	*AUNAME*
1400-00	Пащенко 0.	Введение в технологию АТМ	Деманж Мишель
1400-00	Пащенко 0.	Введение в технологию АТМ	Буассо Марк
1400-00	Пащенко 0.	Введение в технологию АТМ	Мюнье Жан-Мари
1273-92	Гребенкина Н.	The history of England. Absolute Monarchy	Бурова И.И.
1267-89	Арцибашев С.	Тесты. Сборник 11 класс. Варианты и ответы государственного тестирования. Пособие для подготовки к тестированию	без авторов
1267-89	Арцибашев С.	Математические вопросы динамики вязкой несжимаемой жидкости	Ладыжинская Ольга Александровна
1369-99	Светлова В.	Тайна	Хмелевская Иоанна
1267-89	Арцибашев С.	Тайна	Хмелевская Иоанна

Теперь, правда, каждая книга повторяется столько раз, сколько у нее авторов. Поскольку на каждого автора есть отдельная строка, то обойти это просто так невозможно, разве что ограничиться только первым автором в списке:

Пример 3.8

```
SELECT a.rdnumb, a.rdname, b.booknm, min(c.AUNAME)
   FROM TREADER a, TBOOK b, TBOOK_READER ab, TAUTHOR c,
        TBOOK_AUTHOR bc
where ab.reader=a.unikey and ab.bookkey=b.unikey and
      b.unikey= bc.bookkey and bc.author=c.author
GROUP BY a.rdnumb, a.rdname, b.booknm
```

Таблица 3.8. *Выборка списка читателей и взятых ими книг с указанием первого автора*

RDNUMB	*RDNAME*	*BOOKNM*	*MIN*
1267-89	Арцибашев С.	Математические вопросы динамики вязкой несжимаемой жидкости	Ладыжинская Ольга Александровна
1267-89	Арцибашев С.	Тайна	Хмелевская Иоанна
1267-89	Арцибашев С.	Тесты. Сборник 11 класс. Варианты и ответы государственного тестирования. Пособие для подготовки к тестированию	без авторов
1273-92	Гребенкина Н.	The history of England. Absolute Monarchy	Бурова И.И.
1369-99	Светлова В.	Тайна	Хмелевская Иоанна
1400-00	Пащенко О.	Word 6 for Windows	Фаненштих Клаус
1400-00	Пащенко О.	Введение в технологию ATM	Буассо Марк
1456-00	Стародуб Е.	ЯзыкС++	Подбельский Вадим Валериевич

Использование функции **MIN** позволило нам выбирать первого в алфавитном списке автора из группы. Группировка задается перечнем полей, не охватываемых агрегатными функциями, которые необходимо перечислить в опции **GROUP BY.**

Перейдем теперь к более систематическому рассмотрению команды SELECT.

SELECT ПО ОТДЕЛЬНОЙ ТАБЛИЦЕ. ЗАДАНИЕ ВЫБИРАЕМЫХ ПОЛЕЙ

Рассмотрим подробнее выборку данных из отдельной таблицы:

```
SELECT [DISTINCT | ALL] {* | LIST_<val>}
FROM <tableref>
```

Символ * задает выборку всех полей таблицы. Примером может служить уже рассмотренная конструкция примера 3.1:

```
SELECT * FROM TREADER;
```

Результат ее работы приведен в табл. 3.1.

Для задания конкретного перечня столбцов их необходимо задать явно, например, как в примере 3.2:

```
SELECT RDNUMB ,RDNAME FROM TREADER;
```

Результат ее работы приведен в табл. 3.2.

Теперь несколько слов о параметрах запроса DISTINCT и ALL.

Значение ALL (если не указано ничего, то принимается ALL) означает, что будут выбраны все строки таблицы, удовлетворяющие условиям выборки. Об условиях выборки мы поговорим чуть позже.

Значение DISTINCT означает, что будут выбраны только различные строки таблицы. Рассмотрим пример выборки из таблицы читателей, имеющих на руках книги, с параметрами DISTINCT и ALL. Выбирать будем их индивидуальные коды (для выбора фамилий обойтись одной таблицей нельзя, такую выборку мы рассмотрим чуть дальше).

Пример 3.9

```
select ALL reader from TBOOK_READER
select DISTINCT reader from TBOOK_READER
```

Таблица 3.9. *Выборка списка читателей с условиями ALL и DISTINCT*

select ALL reader from TBOOK_READER	***select DISTINCT reader from TBOOK_READER***
READER	READER
36	36
36	37
36	38
37	39
38	40
39	
40	
40	

УСЛОВИЯ ВЫБОРКИ

Условия выборки задаются конструкцией **WHERE <search_condition>].**

```
<search_condition> ::= {<val> <operator>
{ <val> | ( <select_one>)}
| <val> [NOT] BETWEEN <val> AMD <val>
| <val> [NOT] LIKE <val> [ESCAPE <val>]
| <val> [NOT] IN (LIST_<val> | <select_list>)
| <val> IS [NOT] NULL
| <val> { [NOT] {= | < | >} | >= | <=}
{ALL | SOME | ANY} (<select_list>)
| EXISTS ( <select_expr>)
| SINGULAR ( <select_expr>)
| <val> [NOT] CONTAINING <val>
| <val> [NOT] STARTING [WITH] <val>
| ( <search_condition>)
| NOT <search_condition>
| <search_condition> OR <search_condition>
| <search_condition> AND <search_condition>}
```

```
Первый случай: <val> <operator> <val>
<operator> ::={= | < | > | <= | >= | !< | !> | <> | | <> |
!=})
```

Пример 3.10

```
select UNIKEY, BOOKNM from TBOOK where MATHERKEY=5;
```

Таблица 3.10. *Список книг - беллетристики*

UNIKEY	*BOOKNM*
17	Кровь нерожденных
18	Тайна

В ряде случаев одного условия оказывается недостаточно, тогда можно записать составное условие, используя логические операции AND, OR, NOT и при необходимости скобки. Отметим, что этих трех операций достаточно для задания любого логического выражения.

Например, выберем всех читателей, чья фамилия лежит в диапазоне от А до Н.

Пример 3.11

```
select RDNUMB, RDNAME from TREADER
 where RDNAME>='A' AND RDNAME<'0';
```

Таблица 3.11. *Список читателей, чья фамилия лежит в диапазоне от А до Н*

RDNUMB	*RDNAME*
1267-89	Арцибашев С.
1273-92	Гребенкина Н.
1401-99	Грамотный Н.Е.

Другой случай: <val> <operator> (<select_one>) аналогичен первому, только здесь вместо константы, имени столбца или выражения стоит оператор select, обеспечивающий выборку единственного выражения, которое и участвует в сравнении. В конструкции select_one используем агрегатную функцию **SUM**, вычисляющую сумму выражений в скобках по таблице.

Пример 3.12

```
select UNIKEY, BOOKNM from TBOOK
 where 2<(select SUM(BNUMBER) from TBOOK_PLACE where
BOOKKEY=TBOOK.UNIKEY);
```

Таблица 3.12. *Список наименований книг и их ключей, которые имеются более чем в двух экземплярах*

UNIKEY	*BOOKNM*
17	Кровь нерожденных
18	Тайна

Третий случай: *<val>* **[NOT]** **BETWEEN** *<vall>* **AND** *<val2>* полностью эквивалентен конструкции **NOT**[(*<val>* >= *<val1>* **AND** *<val>* <= *<val2>)*.

Например, с помощью BETWEEN пример 3.11 можно переписать в виде

Пример 3.11-1

```
select RDNUMB, RDNAME from TREADER
 where RDNAME BETWEEN  'A' AND 'H';
```

Четвертый случай: <val1> NOT LIKE <val2> /ESCAPE <val3>/ обеспечивает контекстный поиск в символьных выражениях. Конструкция <val1> LIKE <val2> принимает значение ***истина***, если текст <val2> идентичен в определенном смысле тексту <val1>. Для задания текста в <val2> можно использовать помимо обычных символов и символы-заполнители "%" и "_". Символ "_" заменяет любой единичный символ, символ "%" заменяет любое число символов.

'Жил был у бабушки козел Go_home' LIKE 'ил' - ***ложь***, тексты не совпадают.

'Жил был у бабушки козел Go_home' LIKE '%ил%' - ***истина***, тексты с учетом заполнителя "%" совпадают.

'Жил был у бабушки козел Go_home' LIKE '_ил%' - ***истина***, тексты с учетом заполнителей "%" и "_" совпадают.

'Жил был у бабушки козел Go_home' LIKE '%ыл%' - ***истина***, тексты с учетом заполнителя "%" совпадают.

'Жил был у бабушки козел Go_home' LIKE '_ыл%' - ***ложь***, тексты с учетом заполнителей "%" и "_" не совпадают ("_" заменяет только первый символ, в данном случае "Ж").

'Жил был у бабушки козел Go_home' LIKE '%ил%_ыл%' - ***истина***, тексты с учетом заполнителей "%" и "_" совпадают.

Проблема с таким поиском может возникнуть только в том случае, если исходный текст содержит сами заполнители. В нашем примере символ-заполнитель содержится в имени уважаемого козла - "Go_home". Чтобы обойти эту проблему, в шаблоне соответствия <val2> необходимо уметь отличать искомые символы от символов заполнителей. Для этого можно задать управляющий символ, который бы не участвовал в поиске, а только помечал, что следующий за ним символ является не заполнителем, а поисковым. Это обеспечивается конструкцией ESCAPE. Так

<val1> LIKE '%_%' - ***истина*** для любых <val1>.

'Жил был у бабушки козел Go_home' LIKE '%#_%' escape '#' - ***истина***, тексты с учетом заполнителей "%" совпадают (заполнителя "_" здесь нет!).

'Жил был у бабушки козел Go home' LIKE '%#_%' escape '#' - ***ложь***, тексты с учетом заполнителей "%" не совпадают (заполнителя "_" здесь нет, а исходный текст не содержит "_"!).

Пятый случай: <val1> /NOT/ IN (LIST_<val>| <select_list>) задает проверку условия принадлежности <val1> к списку <val2_1>, **[<val2_2>...]**.

Например,

12 in (10, 23, 16) - **ложь**.
12 in (10, 23, 12) - **истина**.

Вместо задания списка явно, его можно задать соответствующей выборкой из базы, используя подзапрос, возвращающий один столбец.

Пример 3.13

```
SELECT rdname, rdnumb
 from treader
 where unikey in
  (select reader from TBOOK_READER where bookkey=18);
```

Таблица 3.13. *Выборка читателей заданной книги*

RDNAME	*RDNUMB*
Арцибашев С.	1267-89
Светлова В.	1369-99

Шестой случай: <val> IS /NOT/ NULL - проверка на пустое значение (не заполнено).

Седьмой случай реализуется с использованием дополнительного подзапроса

```
<val> {[NOT] {= | < | >} | >= | <=} {ALL | SOME | ANY}
(<select_list>)
```

Здесь с заданным выражением сравниваются результаты одностолбцовой выборки.

Конструкция ALL означает, что заданное условие должно выполняться для всех элементов выборки.

Конструкции SOME и ANY являются полными синонимами, пользуйтесь той из них, какая больше нравится. Конструкция принимает значение истина, если заданное условие выполняется хотя бы для одного элемента выборки.

Пример 3.14

```
SELECT a.reader, a.bookkey, r.rdnumb, r.rdname
 from tbook_reader a, treader r
 where a.reader=r.unikey and
 a.bookkey=ANY(SELECT unikey from tbook where matherkey=5);
```

Таблица 3.14. *Список читателей беллетристики*

READER	*BOOKKEY*	*RDNUMB*	*RDNAME*
36	18	1267-89	Арцибашев С.
37	18	1369-99	Светлова В.

Восьмой случай реализуется с использованием дополнительного подзапроса [NOT] EXISTS(<select_expr>). Выражение EXISTS(<select_expr>) принимает значение ***истина***, если в результате выполнения подзапроса находится, по крайней мере, одна строка.

Пример 3.15

```
SELECT BOOKNM, unikey from TBOOK
 where matherkey!=  and
  not EXISTS
  (select * from TBOOK_READER where bookkey=TBOOK.unikey)
```

Таблица 3.15. *Список книг, не востребованных читателями*

BOOKNM	*UNIKEY*
Макрокоманды MS Word	6
Введение в C++ Builder	9
Borland-Технологии. SQL-Link InterBase, Paradox for Windows, Delphi	10
C и C++ Справочник	11
Справочник по правописанию и литературной правке	14
Кровь нерожденных	17

Девятый случай реализуется с использованием дополнительного подзапроса [NOT] SINGULAR (<select_expr>). Выражение SINGULAR (<select_expr>) принимает значение ***истина***, если в результате выполнения подзапроса находится в точности одна строка.

Пример 3.16

```
SELECT BOOKNM, unikey from TBOOK
 where matherkey!=0 and
 SINGULAR
  (select * from TBOOK_READER where bookkey=TBOOK.unikey)
```

Таблица 3.16. *Список книг, имеющих единственного читателя*

BOOKNM	*UNIKEY*
Word 6 for Windows	*1*
Язык C++	8
Введение в технологию АТМ	12
The history of England. Absolute Monarchy	13
Тесты. Сборник 11 класс. Варианты и ответы государственного тестирования. Пособие для подготовки к тестированию	15
Математические вопросы динамики вязкой несжимаемой жидкости	16

Десятый случай: <vall> /NOT/ CONTAINING <val2> обеспечивает контекстный поиск в символьных выражениях. Конструкция <vall> CONTAINING <val2> принимает значение ***истина***, если текст <val2> содержится в тексте <vall>.

Пример 3.17

```
SELECT BOOKNM,unikey from TBOOK
 where BOOKNM CONTAINING 'C++'
```

Таблица 3.17. *Список книг по* C++ *(содержащих* C++ *в названии)*

BOOKNM	*UNIKEY*
Язык C++	8
Введение в C++ Builder	9
C и C++ Справочник	11

Одиннадцатый случай: <vall> NOT STARTING /WITH/ <val2> также обеспечивает контекстный поиск в символьных выражениях. Конструкция <vall> STARTING /WITH/ <val2> принимает значение ***истина***, если текст <vall> начинается с текста <val2>.

Пример 3.18

```
SELECT BOOKNM, unikey from TBOOK
 where BOOKNM STARTING  WITH 'Ma'
```

Таблица 3.18. *Список книг с названием, начинающимся с "Ма"*

BOOKNM	*UNIKEY*
Математика	4
Макрокоманды MS Word	6
Математические вопросы динамики вязкой несжимаемой жидкости	16

ПРЕОБРАЗОВАНИЕ ДАННЫХ ПРИ ВЫБОРКЕ

В ряде случаев при выборке данных из базы с ними необходимо произвести некоторые преобразования.

Функция **CAST** обеспечивает преобразование данных к явно указанному типу:

```
CAST (<val> AS <datatype>),
```

где <val> - преобразуемое выражение, <datatype> - явно указываемый тип данных.

Подробнее типы данных мы обсудим в главах, связанных с описанием данных, а пока приведем просто их перечень.

В InterBase используются следующие типы данных:

```
SMALLINT
INTEGER
FLOAT
DOUBLE PRECISION
DECIMAL(precision [, scale])
NUMERIC(precision [, scale])
DATE
CHAR[ (1...32767) ] или CHARACTER[(1…32767)]
CHARACTER VARYING [ (1...32767) ] или VARCHAR
NCHAR [VARYING] [ (1...32767) ] или NATIONAL CHARACTER
[VARYING] [ (1...32767) ] или NATIONAL CHAR [VARYING]
[ (1...32767) ]
BLOB [SUB_TYPE { int | subtype_name}] [SEGMENT SIZE int]
BLOB [( seglen [, subtype])]
```

Проиллюстрируем использование функции CAST следующим примером, являющимся, может быть, несколько странной модификацией предыдущего примера.

Пример 3.19

```
SELECT BOOKNM,CAST(unikey as DOUBLE PRECISION)*
 3.1415926535 from TBOOK
 where BOOKNM STARTING WITH 'Ma'
```

Таблица 3.19. *Использование функции CAST*

BOOKNM	*F*
Математика	12,566
Макрокоманды MS Word	18,850
Математические вопросы динамики вязкой несжимаемой жидкости	50,265

Вообще функция достаточно полезна везде, где используемые данные должны быть строго определенного типа. При преобразовании необходимо, конечно, помнить, что, во-первых, не все преобразования возможны в принципе и, во-вторых, при преобразованиях возможна потеря или искажение информации. Дело не в ошибках преобразований, а в том, например, что при преобразовании вещественного числа в целое мы обязательно потеряем дробную часть, при преобразовании текста из 100 символов в текст из 60, мы потеряем последние 40 символов. Возможно, это именно то, что мы и хотим, а возможно, и нет. Так что прежде чем проводить преобразования, всегда стоит немножко подумать, зачем мы это делаем и к каким искажениям данных это может привести.

Функция **UPPER** обеспечивает перевод текста в режим только прописных букв (верхний регистр). Последнее бывает важно при операциях сравнения текстов, сортировках и тому подобных случаях, когда нам не нужно различать данные, набранные в разных регистрах. При этом нужно помнить, что стандартная латынь всегда обрабатывается корректно, а для обработки букв национальных алфавитов необходимо явно указывать используемую кодировку (см. CHARACTER SET и COLLATE).

Формат UPPER:

```
UPPER ( <val>)
```

<val> - символьное выражение.

Рассмотрим пример работы функции UPPER, когда национальная кодировка не указана.

Пример 3.20

Таблица 3.20. *Использование функции UPPER*

SELECT BOOKNM from TBOOK where unikey=6 or unikey=7;	*SELECT upper(BOOKNM) from TBOOK where unikey=6 or unikey=7;*
Макрокоманды MS Word Word 6 for Windows	Макрокоманды MS WORD WORD 6 FOR WINDOWS

Из таблицы видно, что латынь перешла из нижнего регистра в верхний, а кириллица осталась без изменений.

Кроме рассмотренных выше функций в операторе SELECT может быть также использована функция GEN_ID, а также дополнительно подключенные пользовательские функции (User Defined). Они могут быть использованы везде, где по синтаксису оператора предполагается конструкция <val>. Применение функции GEN_ID мы рассмотрим подробнее в разделе о триггерах, а пользовательские - в разделе, посвященном подготовке и использованию пользовательских функций.

АГРЕГИРОВАНИЕ ДАННЫХ ПРИ ВЫБОРКЕ

При обработке больших объемов информации часто интересуют не детальные данные, а некоторые их обобщения, такие как суммарные показатели, средние, минимальные и т.п.

Нечто подобное уже встречалось в примерах 3.8 и 3.12.

Прежде всего приведем перечень агрегатных функций:

```
COUNT (* | [ALL] <val> | DISTINCT <val>)
SUM ( [ALL] <val> | DISTINCT <val>)
AVG ( [ALL] <val> | DISTINCT <val>)
MAX ( [ALL] <val> | DISTINCT <val>)
MIN ( [ALL] <val> | DISTINCT <val>)
```

Функция **COUNT** подсчитывает количество строк, удовлетворяющих условиям запроса.

"*" подсчитывает количество строк, удовлетворяющих условиям запроса, включая NULL величины (подсчет не привязан к значениям конкретного поля).

ALL <val> или <val> (параметр ALL предполагается по умолчанию) подсчитывает количество строк, удовлетворяющих условиям запроса. Если таких строк нет, то возвращается NULL.

DISTINCT <val> подсчитывает количество строк, удовлетворяющих условиям запроса, в которых указанное в <val> выражение принимает различные значения.

Подсчитаем количество читателей в нашей библиотеке

Пример 3.21

```
select COUNT(ALL UNIKEY) FROM TREADER;
```

или

```
select COUNT(*) FROM TREADER;
```

или

```
select COUNT(DISTINCT UNIKEY) FROM TREADER;
```

Всё равно, что применить, поскольку строки в таблице TREADER различны.

Результат будет 6.

Другое дело, если нас интересуют читатели, взявшие книги.

Пример 3.22

```
select COUNT(*) FROM TBOOk_READER;
```

Результат будет 8, и он отражает общее количество книг на руках безотносительно к числу читателей и к тому, что может быть выдано несколько экземпляров одной и той же книги.

Пример 3.23

```
select COUNT(DISTINCT READER) FROM TBOOk_READER;
```

Результат будет 5, и он отражает общее количество читателей, имеющих на руках книги.

Пример 3.24

```
select COUNT(DISTINCT BOOKKEY) FROM TBOOk_READER;
```

Результат будет 7, и он отражает общее количество различных книг, выданных читателям.

Функция **SUM** подсчитывает сумму значений указанного выражения по строкам, удовлетворяющим условиям запроса.

ALL <val> или <val> (параметр ALL предполагается по умолчанию) подсчитывает сумму выражений <val> по строкам, удовлетворяющим условиям запроса.

DISTINCT <val> подсчитывает сумму выражений <val> по строкам, удовлетворяющим условиям запроса, в которых указанное в <val> выражение принимает различные значения. Последний вариант, конечно несколько экзотичен, но, может быть, Вам удастся найти для него достойное применение.

Рассмотрим модификацию примера 3.21, заменив COUNT(*) на SUM(1):

Пример 3.25

```
select SUM(1) FROM TREADER;
```

Результат будет 6.

Посмотрим, что будет, если использовать режим DISTINCT.

Пример 3.26

```
select SUM(DISTINCT 1) FROM TREADER;
```

Результат будет 1!

А что если к суммируемому полю добавить другие поля? Попробуем.

Пример 3.27

```
select SUM(1), RDNAME FROM TREADER;
```

Результат: *Dynamic SQL Error. SQL error code = -104. invalid column reference.*

Это несколько не то, что хотелось бы получить, но что, собственно, мы ожидали. SUM должна дать одно значение на группу строк, поле RDNAME - одно значение на каждую строку, а это не одно и то же.

Можно ли вообще получить в одном запросе суммарные или какие-либо другие агрегатные данные вместе с обычными данными. Да, но эти данные должны относиться ко всей группе и порядок группировки должен быть объявлен. Описание группировки рассмотрим чуть дальше, тем более что он относится ко всем агрегатным функциям.

Функция **AVG** подсчитывает среднее значение указанного выражения по строкам, удовлетворяющим условиям запроса.

ALL <val> или <val> (параметр ALL предполагается по умолчанию) подсчитывает среднее значение выражения <val> по строкам, удовлетворяющим условиям запроса, в которых указанное в <val> выражение отлично от NULL.

DISTINCT <val> подсчитывает среднее значение выражения <val> по строкам, удовлетворяющим условиям запроса, в которых указанное в <val> выражение принимает различные значения.

Подсчитаем среднее количество экземпляров книг в нашей библиотеке.

Управление доступом в InterBase на основе SQL 55

Пример 3.28

```
SELECT AVG(BNUMBER) FROM TBOOK_PLACE;
```

Результат будет 2. Число подозрительно "круглое". Дело в том, что функция AVG дает результат того же типа, что и величины, участвующие в суммировании, а они в нашем случае - целые, поэтому наш результат является следствием округления подлинного среднего. Для получения точного результата необходимо явно указать тип обрабатываемых величин. Для указания преобразования используется функция CAST. Ее формат: CAST(<val> AS <тип>), <val> задает преобразуемое выражение, <тип> - тип к которому нужно преобразовать выражение. Подробнее формат CAST и типы данных мы рассмотрим позже, а пока рассмотрим соответствующую модификацию примера 3.28.

Пример 3.29

```
SELECT AVG(CAST(BNUMBER as double precision)) FROM
TBOOK_PLACE;
```

Результат будет 1,84615384615385.

А теперь то же самое с опцией DISTINCT.

Пример 3.30

```
SELECT AVG(DISTINCT CAST(BNUMBER as double precision))
 FROM TBOOK_PLACE;
```

Результат будет 2. Отметим, что «это 2» - не «то 2», что в примере 3.28, а отражение того факта, что число экземпляров в нашей библиотеки колеблется от 1 до 3, а (1+2+3)/3=2.

И еще одно замечание. Вместо AVG(x) можно использовать SUM(x)/COUNT(*), с той лишь разницей, что здесь нет необходимости в приведении типов.

Пример 3.31

```
SELECT SUM(BNUMBER)/COUNT(BNUMBER) FROM TBOOK_PLACE;
```

Результат будет 1,84615384615385.

~~Пример 3.32~~

```
SELECT SUM(DISTINCT BNUMBER)/COUNT(DISTINCT BNUMBER)
 FROM TBOOK_PLACE;
```

Результат будет 2.

Функции **MAX** и **MIN** подсчитывают соответственно максимальное и минимальное значения указанного выражения по строкам, удовлетворяющим условиям запроса.

ALL <val> или <val> (параметр ALL предполагается по умолчанию) подсчитывает максимальное или минимальное значение выражения <val> по строкам, удовлетворяющим условиям запроса, в которых указанное в <val> выражение отлично от NULL.

DISTINCT <val> подсчитывает максимальное или минимальное значение выражения <val> по строкам, удовлетворяющим условиям запроса, исключая дубли величин при выборке максимального или минимального значения.

Разницы между ALL и DISTINCT в данном контексте нет никакой, тем не менее, в документации их описания выделены. Если Вы сумеете понять почему, напишите мне, не хочется оставаться недоумком.

Обобщим примеры 3.29–3.31, выбирая в запросе среднее количество экземпляров книг в нашей библиотеке, общее число книг, а также минимальное и максимальное количество экземпляров одной книги.

Пример 3.33

```
SELECT SUM(BNUMBER), SUM(BNUMBER)/COUNT(BNUMBER),
  MAX(BNUMBER), MIN(BNUMBER)
FROM TBOOK_PLACE;
```

Таблица 3.21. *Характеристики книгохранилища*

SUM	*F_1*	*MAX*	*MIN*
24	1,84615384615385	3	1

SELECT ПО НЕСКОЛЬКИМ ТАБЛИЦАМ. СПОСОБЫ ОБЪЕДИНЕНИЯ ДАННЫХ ИЗ РАЗНЫХ ТАБЛИЦ. ВНЕШНИЕ И ВНУТРЕННИЕ ОБЪЕДИНЕНИЯ

Очень часто бывает необходимо в одном запросе получить данные, которые хранятся в различных таблицах.

Для этого достаточно просто перечислить интересующие нас столбцы из разных таблиц и указать список этих таблиц. Учитывая, что имена столбцов в разных таблицах могут повторяться, в списке столбцов следует указывать, из какой именно таблицы их надо выбирать.

В примере 3.5

```
SELECT * FROM TREADER, TBOOKS;
```

приведена выборка всех данных из двух таблиц. Однако в ней вся информация свалена в кучу. Данные о книгах и читателях никак не связаны.

Управление доступом в InterBase на основе SQL 57

Чтобы увязать их, необходимо указать способ объединения. В данном случае это можно сделать, используя данные таблицы TBOOKREADER, строки которой связывают читателя со взятой им книгой. С другой стороны данные о читателях (таблица TREADER) можно связать с TBOOKREADER по значениям кода читателя (READER в таблице TBOOKREADER и UNIKEY в таблице TREADER), а данные о книгах - по коду книги (BOOKKEY в таблице TBOOK_READER и UNIKEY в таблице TBOOK).

Связь можно задать, используя конструкцию WHERE. Поскольку имена столбцов в таблицах повторяются, то в запросе нужно явно указать, из какой таблицы они выбираются. Запрос в этом случае примет вид примера 3.6:

```
SELECT TREADER.rdnumb, TREADER.rdname, TBOOK.booknm
 FROM TREADER, TBOOK, TBOOK_READER
 where  tbook_reader.reader=treader.unikey and
                tbook_reader.bookkey=tbook.unikey;
```

Вместо довольно длинного имени таблицы можно указать ее алиас (псевдоним). При этом, конечно, нужно задать сам алиас. Задание алиаса таблицы осуществляется совсем просто: после имени таблицы в списке FROM через пробел указывается алиас.

Сразу отметим, что псевдоним можно задавать не только для таблиц, но и для отдельных выбираемых столбцов. Алиас столбца задается точно так же: после выражения для столбца через пробел указывается его алиас. Алиасы столбцов необходимы, прежде всего, для именования вычисляемых столбцов и для различения одноименных столбцов, выбираемых из разных таблиц.

Таким образом, тот же самый запрос можно переписать в следующем виде:

```
SELECT a.rdnumb, a.rdname, b.booknm
 FROM TREADER a, TBOOK b, TBOOK_READER ab
 where  ab.reader=a.unikey and ab.bookkey=b.unikey;
```

Организация связи данных при выборке на основе конструкции WHERE является, пожалуй, основной, но не единственно возможной.

Другим приемом связывания данных является указание способа объединения таблиц в списке <tableref> после конструкции FROM. Формат такого объединения имеет вид

<joined_table> ::= <tableref> <join_type> **JOIN** <tableref>
ON <search_condition> | (<joined_table>)

```
<join-type> ::= { [INNER] | {LEFT | RIGHT | FULL }
[OUTER] } JOIN
```

JOIN ... ON - соединение строк таблиц на основе условия, заданного после ON.

Тип соединения задается ключевым словом INNER или OUTER; если ни одно из них не указано, то принимается INNER.

Тип соединения INNER задает "внутреннее соединение". В таблицах соединяются только те строки, для которых выполняется <search_condition>.

Ключевое слово OUTER (внешний) не является обязательным, оно не используется ни в каких операциях с данными и имеет смысл только в комбинации со спецификацией типа соединения (LEFT, RIGHT, FULL).

Таблицу в соединении будем называть левой, если она стоит перед (слева) ключевым словом JOIN, и правой, если она стоит после (справа) от него.

LEFT (OUTER) - тип соединения "левое (внешнее)". Левое соединение таблиц включает в себя все строки из левой таблицы и те строки из правой таблицы, для которых выполняется <search_condition>. Для строк из левой таблицы, для которых не найдено соответствия в правой, в столбцы, извлекаемые из правой таблицы, заносятся значения NULL.

RIGHT (OUTER) - тип соединения " правое (внешнее)". Правое соединение таблиц включает в себя все строки из правой таблицы и те строки из левой таблицы, для которых выполняется <search_condition>. Для строк из правой таблицы, для которых не найдено соответствия в левой, в столбцы, извлекаемые из левой таблицы, заносятся значения NULL.

FULL (OUTER) - тип соединения "полное (внешнее)". Это комбинация левого и правого соединений. В полное соединение включаются все строки из обеих таблиц. Для совпадающих строк поля заполняются реальными значениями, для несовпадающих строк поля заполняются в соответствии с правилами левого и правого соединений.

При использовании внутреннего соединения конструкции JOIN и WHERE полностью взаимозаменяемы. Использование соединенных таблиц часто облегчает восприятие оператора SELECT, особенно, когда используется естественное соединение. Если не использовать соединенные таблицы, то при выборе данных из нескольких таблиц необходимо явно указывать условия соединения в разделе WHERE. Если при этом пользователь указывает сложные критерии отбора строк, то в разделе WHERE смешиваются такие семантически различные понятия, как условия связи таблиц, так и условия отбора строк. Каких-либо иных дополнительных преимуществ конструкция JOIN перед конструкцией WHERE не имеет, так что выбор представляется делом вкуса.

Перепишем запрос из примера 3.6. с использованием конструкции JOIN.

Пример 3.34

```
SELECT a.rdnumb, a.rdname, b.booknm
 FROM TREADER a JOIN TBOOK b on EXISTS(SELECT * FROM
TBOOK_READER ab
 where ab.reader=a.unikey and ab.bookkey=b.unikey);
```

Таблица. 3.22. *Выборка списка читателей и взятых ими книг*

RDNUMB	*RDNAME*	*BOOKNM*
1400-00	Пащенко О.	Word 6 for Windows
1456-00	Стародуб Е.	Язык С++
1400-00	Пащенко О.	Введение в технологию АТМ
1273-92	Гребенкина Н.	The history of England. Absolute Monarchy
1267-89	Арцибашев С.	Тесты. Сборник 11 класс. Варианты и ответы государственного тестирования. Пособие для подготовки к тестированию
1267-89	Арцибашев С.	Математические вопросы динамики вязкой несжимаемой жидкости
1267-89	Арцибашев С.	Тайна
1369-99	Светлова В.	Тайна

А теперь проделаем то же самое, но используя внешние соединения.

Пример 3.35

```
SELECT a.rdnumb, a.rdname, b.booknm
 FROM TREADER a LEFT JOIN TBOOK b on
 EXISTS(SELECT * FROM TBOOK_READER ab
 where ab.reader=a.unikey and ab.bookkey=b.unikey);
```

Таблица 3.23. *Выборка списка всех читателей и взятых ими книг*

RDNUMB	*RDNAME*	*BOOKNM*
1267-89	Арцибашев С.	Тесты. Сборник 11 класс. Варианты и ответы государственного тестирования. Пособие для подготовки к тестированию

RDNUMB	RDNAME	BOOKNM
1267-89	Арцибашев С.	Математические вопросы динамики вязкой несжимаемой жидкости
1267-89	Арцибашев С.	Тайна
1369-99	Светлова В.	Тайна
1456-00	Стародуб Е.	Язык С++
1273-92	Гребенкина Н.	The history of England. Absolute Monarchy
1400-00	Пащенко О.	Word 6 for Windows
1400-00	Пащенко О.	Введение в технологию ATM
1401-99	*Грамотный Н.Е.*	

Пример 3.36

```
SELECT a.rdnumb, a.rdname, b.booknm
 FROM TREADER a RIGHT JOIN TBOOK b on
 EXISTS(SELECT * FROM TBOOK_READER ab
 where ab.reader=a.unikey and ab.bookkey=b.unikey);
```

Таблица 3.24. *Выборка списка читателей и всех книг*

RDNUMB	RDNAME	BOOKNM
		Программирование
		Учебники
		Математика
		Беллетристика
		Макрокоманды MS Word
1400-00	Пащенко О.	Word 6 for Windows
1456-00	Стародуб Е.	Язык С++
		Введение в С++ Builder
		Borland-Технологии. SQL-Link InterBase, Paradox for Windows. Delphi
		С и С++ Справочник

RDNUMB	*RDNAME*	*BOOKNM*
1400-00	Пащенко О.	Введение в технологию АТМ
1273-92	Гребенкина Н.	The history of England. Absolute Monarchy
		Справочник по правописанию и литературной правке
1267-89	Арцибашев С.	Тесты. Сборник 11 класс. Варианты и ответы государственного тестирования. Пособие для подготовки к тестированию
1267-89	Арцибашев С.	Математические вопросы динамики вязкой несжимаемой жидкости
		Кровь нерожденных
1267-89	Арцибашев С.	Тайна
1369-99	Светлова В.	Тайна

Пример 3.37

```
SELECT a.rdnumb, a.rdname, b.booknm
 FROM TREADER a FULL JOIN TBOOK b on
 EXISTS(SELECT * FROM TBOOK_READER ab
 where ab.reader=a.unikey and ab.bookkey=b.unikey);
```

Таблица 3.25. *Выборка списка всех читателей и всех книг*

RDNUMB	*RDNAME*	*BOOKNM*
		Программирование
		Учебники
		Математика
		Беллетристика
		Макрокоманды MS Word
1400-00	Пащенко О.	Word 6 for Windows
1456-00	Стародуб Е.	Язык С++
		Введение в С++ Builder
		Borland-Технологии. SQL-Link InterBase. Paradox for Windows. Delphi

RDNUMB	RDNAME	BOOKNM
		С и С++ Справочник
1400-00	Пащенко О.	Введение в технологию АТМ
1273-92	Гребенкина Н.	The history of England. Absolute Monarchy
		Справочник по правописанию и литературной правке
1267-89	Арцибашев С.	Тесты. Сборник 11 класс. Варианты и ответы государственного тестирования. Пособие для подготовки к тестированию
1267-89	Арцибашев С.	Математические вопросы динамики вязкой несжимаемой жидкости
		Кровь нерожденных
1267-89	Арцибашев С.	Тайна
1369-99	Светлова В.	Тайна
1401-99	*Грамотный Н.Е.*	

В табл. 3.23-3.25 выделены: курсивом строки, появившиеся дополнительно за счет использования левого соединения, подчеркиванием - правого.

Отметим также, что в списке выбираемых столбцов могут быть и такие, которые сами по себе являются результатом выборки с помощью SELECT. В качестве примера рассмотрим следующую несколько экзотическую, но вполне работоспособную конструкцию.

Пример 3.38

```
select UNIKEY,
  (SELECT auname from tauthor
    where tbook_author.author=tauthor.author) auname,
  (SELECT BOOKNM from tbook
   where tbook_author.bookkey=tbook.unikey) bookname
  from tbook_author;
```

Таблица 3.26. *Выборка списка авторов и книг*

UNIKEY	*AUNAME*	*BOOKNAME*
58	Культин Н.Б.	Макрокоманды MS Word
59	Буассо Марк	Введение в технологию ATM
60	Деманж Мишель	Введение в технологию ATM
61	Мюнье Жан-Мари	Введение в технологию ATM
62	Луис Дерк	C и C++ Справочник
63	Дунаев Сергей	Borland-Технологии. SQL-Link InterBase, Paradox for Windows, Delphi
64	Елманова Н.З.	Введение в C++ Builder
65	Кошель С П.	Введение в C++ Builder
66	Подбельский Вадим Валериевич	Язык C++
67	Хаселир Райнер Г.	Word 6 for Windows
68	Фаненштих Клаус	Word 6 for Windows
69	Ладыжинская Ольга Александровна	Математические вопросы динамики вязкой несжимаемой жидкости
70	без авторов	Тесты. Сборник 11 класс. Варианты и ответы государственного тестирования. Пособие для подготовки к тестированию
71	Розенталь Д.Э.	Справочник по правописанию и литературной правке
72	Бурова И.И.	The history of England. Absolute Monarchy
73	Дашкова Полина	Кровь нерожденных
74	Хмелевская Иоанна	Тайна

Следует, правда, заметить, что в данном случае то же самое можно было бы получить заметно проще:

```
select a.UNIKEY, b.auname, c. BOOKNM
 from tbook_author a, tauthor b, tbook c
  where a.author=b.author and a.bookkey=c.unikey
```

ДОПОЛНИТЕЛЬНАЯ ОБРАБОТКА РЕЗУЛЬТАТОВ, ФИЛЬТРАЦИЯ И СОРТИРОВКА

В ряде случаев уже выбранные данные нуждаются в дополнительной обработке. Часть такой обработки можно выполнить в рамках SQL запросов.

Прежде всего, это упорядочение данных. Полученные данные могут быть отсортированы по значениям одного или нескольких столбцов запроса. Сортировка задается конструкцией ORDER BY:

```
[ORDER BY <order_list>]
<order_list> =
{col | int} [COLLATE collation] [ASC[ENDING] |
DESC[ENDING]]
[, <order_list>].
```

Как видно из описания синтаксиса, в списке упорядочения могут быть как имена столбцов, так и их порядковые номера (но не выражения или алиасы).

Пример 3.39

```
SELECT unikey, bnumber n1, bnumber*bnumber n2
 FROM TBOOK_PLACE
ORDER BY n1;
```

Результат- "Dynamic SQL Error. SQL error code = -206. Column unknown. N1".

А если указать вместо алиаса имя столбца, то все будет в порядке.

Пример 3.40

```
SELECT unikey, bnumber n1, bnumber*bnumber n2
 FROM TBOOK_PLACE
ORDER BY bnumber;
```

Таблица 3.27. *Тестовая выборка №1 из TBOOK_PLACE*

UNIKEY	*N1*	*N2*
47	1	1
49	1	1
51	1	1
53	1	1
48	2	4
50	2	4
52	2	4
54	2	4
55	2	4
56	2	4
57	2	4
45	3	9
46	3	9

А как отсортировать по столбцу N2, которому не соответствует ни одно поле таблицы? В этом случае единственный возможный путь - явно указать номер столбца в списке выборки. В нашем случае - 3. Поскольку одному значению поля N2 соответствует несколько строк, то для обеспечения однозначности выборки упорядочим ее дополнительно по убыванию значений первого столбца.

Пример 3.41

```
SELECT unikey, bnumber n1, bnumber*bnumber n2
 FROM TBOOK_PLACE
ORDER BY 3, 1 DESC;
```

или

```
SELECT unikey, bnumber n1, bnumber*bnumber n2
 FROM TBOOK_PLACE
ORDER BY 3, UNIKEY DESC;
```

Конструкция ASC[ENDING], DESC[ENDING] указывает на вид сортировки: по возрастанию или убыванию соответственно. Если не указано ничего, то принимается ASC.

Таблица 3.28. *Тестовая выборка №2 из TBOOK_PLACE*

UNIKEY	*N1*	*N2*
47	1	1
49	1	1
51	1	1
53	1	1
48	2	4
50	2	4
52	2	4
54	2	4
55	2	4
56	2	4
57	2	4
45	3	9
46	3	9

Сортировка, конечно, не является единственным видом обработки результатов. Мы уже рассматривали получение суммарных данных в запросах на основе применения агрегатных функций. При этом мы получали единственную строку итогов по базе, однако в ряде случаев необходимо провести агрегирование данных по заданной группе признаков. Например, это могут быть данные о суммарных продажах по филиалам торговой фирмы, объемы производства по цехам, по месяцам года и тому подобное. Таких примеров можно привести множество. Для реализации подобных задач применяется конструкция GROUP BY:

[GROUP BY col **[COLLATE** collation] [, col **[COLLATE** collation] **...]**

Рассмотрим модификацию примера 3.27, который у нас не получился. Для этого включим в него группировку по читателям.

```
select SUM(1), RDNAME FROM TREADER GROUP BY RDNAME;
```

Получим список читателей и единичек. Запрос сработал, но результат малоинтересен.

Теперь получим такой же список, но по каждому читателю выдадим количество книг, которые он взял.

Пример 3.42

```
select a.RDNAME, SUM(1)
 FROM TREADER a, TBOOK_READER b
 where (a.UNIKEY=b.READER) GROUP BY RDNAME;
```

Таблица 3.29. *Список читателей с указанием количества взятых ими книг*

RDNAME	*SUM*
Арцибашев С.	3
Гребенкина Н.	1
Пащенко О.	2
Светлова В.	1
Стародуб Е.	1

Отметим, что в списке выборки могут находиться только агрегируемые выражения и выражения, перечисленные в списке GROUP BY.

Следующая конструкция HAVING <search_condition> может показаться излишней. В самом деле, фильтрация данных прекрасно выполняется с помощью конструкции WHERE. В чем отличие конструкции HAVING от WHERE? Конструкция WHERE применяется к каждой строке и определяет, будет ли она включена в выборку или нет. Конструкция HAVING применяется к уже полученной выборке и фильтрует ее. В большинстве случаев последнее просто неэффективно, но не всегда, поскольку некоторые данные в процессе выборки просто неизвестны. Например, в случае запроса в примере 3.42 мы не можем отбросить читателей, взявших менее 2 книг, поскольку пока расчет не выполнен, это просто неизвестно. Но когда он уже сделан, такой фильтр легко задать, используя HAVING.

Пример 3.43

```
select a.RDNAME, SUM(1) sumbook
 FROM TREADER a, TBOOK_READER b
 where (a.UNIKEY=b.READER) GROUP BY RDNAME
HAVING SUM(1)>1;
```

Таблица 3.30. *Список читателей, взявших более 1 книги с указанием количества взятых ими книг*

RDNAME	*SUMBOOK*
Арцибашев С.	3
Пащенко О.	2

ОБЪЕДИНЕНИЕ НЕСКОЛЬКИХ ОДНОТИПНЫХ ЗАПРОСОВ В ОДИН

UNION *[ALL]* объединяет результаты выборки из двух или более таблиц, которые идентичны по структуре (результаты выборки, а не таблицы; сами таблицы могут иметь и разную структуру). Если используется конструкция ALL, то в выборке могут присутствовать и одинаковые строки, полученные из разных таблиц.

Если поля или выражения, выбираемые из разных таблиц и помещаемые в один столбец, имеют разные имена, то им необходимо дать псевдоним.

Пример 3.44

```
select RDNAME, CAST("Активный читатель" as VARCHAR(20))
RD_Active
 FROM TREADER a
 WHERE EXISTS(
  select count(1) from TBOOK_READER b where
b.READER=a.UNIKEY
     HAVING count(1)>2)
UNION
select RDNAME, CAST("Средний читатель" as VARCHAR(20))
RD_Active
 FROM TREADER a
 WHERE EXISTS(
  select count(1) from TBOOK_READER b where
b.READER=a.UNIKEY
     HAVING count(1)<3 and count(1)>0)
UNION
select RDNAME, CAST("Пассивный читатель" as VARCHAR(20))
RD_Active
 FROM TREADER a
 WHERE
  NOT EXISTS (select * from TBOOK_READER b where
b.READER=a.UNIKEY);
```

Хаблица 3.31. *Список читателей по их активности*

RDNAME	RD_ACTIVE
Арцибашев С.	Активный читатель
Грамотный Н.Е.	Пассивный читатель
Гребенкина Н.	Средний читатель
Пащенко О.	Средний читатель
Светлова В.	Средний читатель
Стародуб Е.	Средний читатель

Замечание

В списке полей выборки могут появляться и отдельные элементы полей - *массивы*.

В то же время использование массивов в полях таблиц баз данных без особой необходимости я бы не рекомендовал. Дело не в том, хорошо или плохо они реализованы в InterBase, просто данный тип не является элементом стандарта SQL и при попытке мигрировать с платформы InterBase на какую-либо иную, а от этого никто не гарантирован - желания клиентов непредсказуемы, может возникнуть масса проблем. Кроме того, данные типа массив не вписываются, вообще говоря, в концепцию реляционных баз, да и массив всегда можно реализовать в виде дополнительной таблицы. С другой стороны, если Вы считаете, что без них Вам будет очень плохо, то пользуйтесь, но не забывайте о возникающих при этом проблемах.

3.2. Добавление данных. Команда INSERT

Назначение и основные возможности команды

Команда **INSERT** предназначена для добавления данных в базу. С ее помощью можно добавить в указанную таблицу или представление одну или сразу несколько строк. Возможность применения операции добавления каким-либо пользователем определяется тем, какие права доступа были ему даны (см. команды GRANT и REVOKE).

Синтаксискоманды Insert

Полный синтаксис команды Insert имеет вид

INSERT [TRANSACTION transaction] **INTO** <object> [(LIST_col)]
{**VALUES** (LIST_<val>) | <select_expr>};

<object> ::= имя_таблицы или имя_ обзора

<val> ::=/

: variable | <constant> | **<expr>**

| <function> | udf (/ LIST_<val>/)

| NULL | USER | RDB$DB_KEY | ?

} /**COLLATE** collation/

<constant> = num |' string' | charsetname ' string '

<expr> = любое допустимое в SQL выражение, имеющее своим результатом единственное значения для столбца

<function>::=/

CAST (<val> **AS <datatype>)**

| **UPPER (**<val>)

| **GEN_ID** (generator, <val>)

}

В версиях InterBase, начиная с 6, допустимо использовать также функцию EXTRACT(part **FROM** <DTExpr>).

part ::= /**DAY** | **HOUR** | **MINUTE** | **MONTH** | **SECOND** | **WEEKDAY | YEAR | YEARDAY**

<DTExpr> ::= Любое корректное SQL выражение, дающее в результате единственное значение типа дата - время.

udf- пользовательская функция

<select_expr> = SELECT, возвращающий несколько строк (возможно 0) со столбцами в том же порядке и того же типа, что и в списке, заданном в конструкции INTO.

Замечание. Если в списке ввода (INTO) перечислены не все столбцы таблицы, то те столбцы, которые не включены в список, получают значения NULL, если для этих полей в описании таблицы не предусмотрено значение по умолчанию.

Поскольку с таблицей могут быть связаны триггеры, являющиеся по своей сути специальными программами, осуществляющими контроль и предварительную обработку данных, то в результате выполнения команды INSERT и работы триггера новые значения могут отличаться от

введенных. Также могут быть и определены значения для столбцов, которые не указаны в перечне INTO.

Отметим, что конструкция ***:variable*** доступна только в процедурах и триггерах базы данных, либо во внедренном SQL в приложениях.

Добавление отдельной строки

При добавлении отдельной строки в списке INTO перечисляются столбцы, в которые вводятся значения, а сами значения задаются в списке VALUES. Вместо списка столбцов можно указать символ "*"; в этом случае предполагается, что вводятся значения всех столбцов таблицы в специфицированном при создании таблицы порядке. Поскольку порядок столбцов может изменяться, то такая конструкция является потенциально опасной и лучше ее не применять.

Рассмотрим добавление строки в таблицу TBOOK.

Пример 3.45

Таблица 3.32. *Содержание TREADER перед вводом (select * from TREADER;)*

UNIKEY	*RDNUMB*	*RDNAME*
36	1267-89	Арцибашев С.
37	1369-99	Светлова В.
38	1456-00	Стародуб Е.
39	1273-92	Гребенкина Н.
40	1400-00	Пащенко О.
83	1401-99	Грамотный Н.Е.

```
INSERT INTO TREADER (RDNUMB, RDNAME)
 VALUES ('1111-98', 'Пугачева А.Б.');
```

Для данной таблицы предусмотрен триггер, поэтому значение для неуказанного в списке столбца UNIKEY сформируется автоматически.

Возможный результат приведен в табл. 3-33. В Вашем случае он может быть и иным в зависимости от состояния генератора SYSNUMBER.

Отметим, что триггер для таблицы TREADER в нашем случае предусматривает обязательность указания полей RDNUMB и RDNAME, поэтому Вам просто не удастся выполнить INSERT без их указания.

Таблица 3.33. *Содержание TREADER после ввода (select * from TREADER;)*

UNIKEY	RDNUMB	RDNAME
36	1267-89	Арцибашев С.
37	1369-99	Светлова В.
38	1456-00	Стародуб Е.
39	1273-92	Гребенкина Н.
40	1400-00	Пащенко О.
83	1401-99	Грамотный Н.Е.
84	1111-98	Пугачева А.Б.

При указании значений необходимо следить за совместимостью типов вводимых данных и типов данных в таблице.

Здесь все время говорилось о команде INSERT применительно к таблицам, однако она точно так же применима и для представлений, но с некоторыми оговорками, которые мы сделаем при рассмотрении представлений.

В данном случае вводимые значения задавались как константы, но вместо них могут стоять и любые допустимые в SQL выражения, возвращающие единственное значение, в том числе и конструкции SELECT.

Добавление группы строк

Одной командой INSERT можно добавить в таблицу и несколько строк. Перечень добавляемых строк в этом случае задается конструкцией SELECT.

В качестве примера рассмотрим ту же таблицу TREADER. Добавим в нее в качестве читателей авторов книг из таблицы TAUTHOR, а номера читательских билетов для них придумаем сами. Например, номер можно сформировать из поля AUTHOR, к которому будем добавлять цифры 00.

Пример 3.46

```
INSERT INTO TREADER (RDNUMB, RDNAME)
 SELECT CAST(AUTHOR as VARCHAR(8)) || '-00', AUNAME FROM
TAUTHOR
  where AUNAME<'Д';
```

Таблица 3.34. *Содержание TREADER после ввода (select * from TREADER;)*

UNIKEY	RDNUMB	RDNAME
36	1267-89	Арцибашев С.
37	1369-99	Светлова В.
38	1456-00	Стародуб Е.
39	1273-92	Гребенкина Н.
40	1400-00	Пащенко О.
83	1401-99	Грамотный Н.Е.
84	1111-98	Пугачева А.Б.
85	22-00	Бурова И.И.
86	33-00	Буассо Марк

3.3. Обновление данных. Команда UPDATE

Назначение и основные возможности команды

Команда **UPDATE** предназначена для изменения всех или части строк в таблице, представлении или курсоре в зависимости от задаваемых условий коррекции. Право на изменение строк таблицы устанавливается для отдельных пользователей командами GRANT и REVOKE.

Базовый синтаксис команды UPDATE

```
UPDATE [TRANSACTION transaction] {table | view}
SET LIST_<sets> [WHERE <search_condition> | WHERE CURRENT
OF cursor];

<sets> ::= col = <val>
```

Здесь мы рассмотрим только первую часть синтаксиса. По второй части лишь отметим, что он обеспечивает изменение одной строки, соответствующей текущей строке курсора. Работа с курсорами возможна

только в процедурах и триггерах базы данных, либо во внедренном SQL в приложениях и будет рассмотрена позже.

```
UPDATE [TRANSACTION transaction] {table | view}
SET LIST_<sets> [WHERE <search_condition>

<val> ::={
col [<array_dim>] | : variable
 | <constant> | <expr> | <function>
 | udf ( [ <val> [, <val> ...]])
 | NULL | USER | ?}
 [COLLATE collation]
```

<Array_dim> ::= [LIST_Dim]

Dim ::= [x:]y

y - задает размерность массива.

x - задает нижнюю границу массива (если задано x:y, то индекс в массиве меняется от x до y)

<constant> ::= num | ' string' | charsetname ' string'

<expr> ::= любое допустимое в SQL выражение, имеющее своим результатом единственное значения для столбца

<function> ::= {
CAST (<val> AS <datatype>)
| **UPPER** (<val>)
| **GEN_ID** (generator, <val>)
}

В версиях InterBase, начиная с 6, допустимо использовать также функцию **EXTRACT**(part **FROM** <DTExpr>).

part ::= **/DAY | HOUR | MINUTE | MONTH | SECOND | WEEKDAY | YEAR | YEARDAY /**

<DTExpr> ::= Любое корректное SQL выражение, дающее в результате единственное значение типа дата - время.

udf - пользовательская функция

<search_condition> ::= { <val> <operator> { <val> | (<select_one>)}

Управление доступом в InterBase на основе SQL 75

| <val> **NOT** BETWEEN <val> **AND** <val>
| <val> *[***NOT***]* LIKE <val> *[*ESCAPE <val>*]*
| <val> *[***NOT***]* IN (SET_<val> | <select_list>)
|<val>IS **NOT** NULL
| <val> *{ [***NOT***] =* | < | >} | >= | <=*}*
{ **ALL** | **SOME** | **ANY***}* (<select_list>)
| **EXISTS** (<select_expr>)
| **SINGULAR** (<select_expr>)
| <val> **NOT** **CONTAINING** <val>
| <val> **NOT** **STARTING** *[***WITH***]* <val>
| (<search_condition>)
| **NOT** <search_condition>
| <search_condition> **OR** <search_condition>
| <search_condition> **AND** <search_condition>*]*

Подробное описание конструкции <search_condition>, задающей условия выборки данных, и соответствующие примеры даны в описании команды SELECT.

Существенное значение в команде UPDATE играет конструкция WHERE, определяющая какие именно строки подлежат изменению. Если она отсутствует, то будут изменены все строки таблицы, то есть значением конструкции WHERE по умолчанию является истина.

Конструкция SET задает перечень изменяемых столбцов и их новых значений.

Как следует из синтаксического описания, новые значения могут быть любым выражением, допустимым в SQL, а также могут задаваться через значения других столбцов той же таблицы.

Конструкция WHERE задает условия обновления. Она полностью идентична одноименной конструкции в команде SELECT, так что здесь нет необходимости в ее подробном описании.

Задание условий обновления. Обновление группы строк

При обновлении данных одной командой можно внести изменения в одну или несколько строк.

Основной режим обновления состоит в изменении значений одного или нескольких столбцов в конкретной строке таблицы. В этом случае конструкция WHERE имеет вид

WHERE <col1>=<val1> [and <col2>=<val2> ...] .

Здесь coll, col2, ... - список полей, образующих уникальный ключ в таблице. Использование уникального ключа гарантирует, что изменению не будут подвергнуты никакие посторонние строки таблицы.

Рассмотрим изменение номера читательского билета у читателя (например, при процедуре перерегистрации). Использование фамилии при подобном действии опасно, поскольку нет никакой гарантии, что у нас нет однофамильцев, а тогда всем им будет приписан один и тот же номер билета, что, конечно же, недопустимо.

Пример 3.47

```
UPDATE TREADER SET RDNUMB='1278-98'
 where UNIKEY=39;
```

В результате выполнения команды изменится значение строки таблицы с кодом 39.

Изменение можно отследить командой SELECT:

```
select *  from TREADER where unikey=39;
```

Соответствующие изменения видны из табл. 3.35.

Таблица 3.35. *Содержание TREADER до и после ввода изменений*

	UNIKEY	*RDNUMB*	*RDNAME*
До ввода изменений	39	1273-92	Гребенкина Н.
После ввода изменений	39	1278-98	Гребенкина Н.

В данном примере изменялось только одно поле. Рассмотрим аналогичное изменение, но с несколькими полями.

```
UPDATE TREADER SET RDNUMB='1278-98',
 RDNAME=  Гребенкина Н.'
where UNIKEY=39;
```

Необходимость в групповых изменениях возникает значительно реже. Более того, часто тот факт, что необходимо ввести сразу много однотипных изменений в базу данных, свидетельствует о том, что данные не были должным образом нормализованы. Тем не менее, такая процедура иногда может быть очень полезна. Например, при изменении срока, на который выдаются книги на месяц, необходимо изменить столбец FIRSTDATE на 30 дней и соответственно изменить столбец NEXTDATE так, чтобы он не превосходил столбец FIRSTDATE во всей таблице. Это можно сделать двумя командами UPDATE.

Управление доступом в InterBase на основе SQL 77

Пример 3.48

```
UPDATE TBOOK_READER SET FIRSTDATE = FIRSTDATE+30;
```

Здесь мы заменили срок сдачи книг по всей таблице.

Теперь заменим дату продления для тех случаев, когда она меньше основной даты:

```
UPDATE TBOOK_READER SET NEXTDATE = FIRSTDATE WHERE NEXTDATE
< FIRSTDATE;
```

3.4. Удаление данных. Команда DELETE

Назначение и основные возможности команды

Команда DELETE удаляет из таблицы одну или несколько строк в зависимости от задаваемых условий удаления. Право на удаление строк из таблицы устанавливается для отдельных пользователей командами GRANT и REVOKE.

Базовый синтаксис команды DELETE

```
DELETE [TRANSACTION transaction] FROM table
{ [WHERE <search_condition>] | WHERE CURRENT OF cursor};
```

Здесь мы рассмотрим только первую часть синтаксиса. По второй части лишь отметим, что он обеспечивает удаление одной строки, соответствующей текущей строке курсора. Работа с курсорами возможна только в процедурах и триггерах базы данных, либо во внедренном SQL в приложениях и будет рассмотрена позже.

Задание условий удаления. Удаление группы строк

Исключительное значение в команде DELETE играет конструкция WHERE, определяющая какие именно строки подлежат удалению. Если она отсутствует, то будут удалены все строки таблицы, то есть значением конструкции WHERE по умолчанию является истина.

При рассмотрении команды INSERT в таблицу TREADER был введен ряд строк. Посмотрим, как их можно удалить. Проще всего это сделать, используя тот факт, что генерируемые при вводе значения первичного ключа UNIKEY возрастают. Наибольшее значение у ранее введенных данных было 83, поэтому можно написать следующую команду:

Пример 3.49

```
DELETE FROM TREADER
  where UNIKEY>83;
```

Такая конструкция может оказаться неудобной, если часть данных из диапазона нужно исключить. В этом случае удобнее задать список значений ключевого поля явно.

Пример 3.50

```
DELETE FROM TREADER
  where UNIKEY in (84,85,86);
```

Глава 4

Описание данных на основе SQL

4.1. Организация данных в InterBase. Типы данных

Данные в реляционных базах данных, в частности в InterBase, хранятся в плоских таблицах. В каждом столбце таблицы хранятся данные одного определенного типа. При этом нужно помнить, что данные, по которым осуществляется поиск, должны быть простого (неструктурированного) типа, для которого определены операции сравнения. Данные других типов допускаются, если можно так сказать, "в порядке исключения" и для работы с ними используются средства, выходящие за пределы стандарта SQL.

Приведем перечень обрабатываемых в InterBase типов данных.

SMALLINT - слово, короткое целое (2-байтовое) со знаком (от -32 768 до 32 767).

INTEGER - двойное слово, длинное целое (4-байтовое) со знаком (от -2 147 483 648 до 2 147 483 647).

FLOAT - числа с плавающей точкой одинарной точности (4 байта) - 7 значащих цифр.

DOUBLE PRECISION - числа с плавающей точкой двойной точности (8 байтов) - 15 значащих цифр.

DECIMAL (размер, точность) / NUMERIC (размер, точность). Размер переменной (от 1 до 15) указывает гарантированную точность переменной, то есть число значащих цифр. Точность (от 1 до 15) задает число цифр после запятой (должно быть меньше или равно размеру). Например, DECIMAL(10,3) содержит числа в формате: ppppppp.sss

Типы данных DECIMAL и NUMERIC имеют смысл только для внешнего представления данных. В базе они реально хранятся в одном из

80 Глава 4

основных числовых форматов (SMALLINT, INTEGER, FLOAT или DOUBLE PRECISION).

- Если размер и точность не указаны, то данные хранятся как INTEGER (от -2 147 483 648 до 2 147 483 647).
- Если точность не указана, то принимается 0. Хранимый тип при этом будет зависеть от размера. Если размер меньше 5, то SMALLINT. Если размер ∈[5, 9] , то INTEGER. Если размер больше 9, то DOUBLE PRECISION.
- Если указаны и размер, и точность, то хранимый тип будет зависеть от введенной величины размера. Если размер меньше 5, то SMALLINT. Если размер ∈[5, 9] , то INTEGER. Если размер больше 9, то DOUBLE PRECISION. То есть, числа типа ppp.ss хранятся, как pppss.

DATE в версии до 6 или TIMESTAMP в версиях от 6 (8 байт) с 1.01.100 до 29.02.32768, включает также данные о времени; DATE в версии от 6 - 4 байта (только дата); TIME в версии от 6 - 4 байта (только время).

CHAR(n) / CHARACTER(n) n символов (от 1 до 32 767 байт) - строка фиксированной длины; максимальная длина - 32K.

CHARACTER VARYING(n) / VARCHAR(n) / CHAR VARYING(n) n символов (от 1 до 32 767 байт) - строка переменной длины; максимальная длина - 32K.

NCHAR(n) / NATIONAL CHARACTER(n) / NATIONAL CHAR(n) n символов (от 1 до 32 767 байт) - строка фиксированной длины, использующая кодовый набор ISO8859_1.

NCHAR VARYING (n) / NATIONAL CHARACTER VARYING (n) / NATIONAL CHAR VARYING (n) n символов (от 1 до 32 767 байт) - строка переменной длины, использующая кодовый набор ISO8859_1.

BLOB [SUB_TYPE { int | subtype_name}] [SEGMENT SIZE int] [CHARACTER SET charname] / BLOB [(seglen [, subtype])].

SUB_TYPE:

- 0 - неструктурированный, обычно используется для двоичных данных или данных неопределенного типа
- 1 - текст
- 2 - двоичное языковое представление BLR (Binary language representation)
- 3 - Access controllist
- 4 - зарезервировано
- 5 - закодированное описание метаданных текущей таблицы
- 6 - описание ненормально завершенной транзакции к нескольким базам
- <0 - пользовательский тип

Описание данных на основе SQL 81

SEGMENT SIZE - размер блока, через который осуществляется чтение-запись данных BLOB в приложениях, использующих embedded SQL.

Все перечисленные типы данных, кроме BLOB, могут быть организованы в массивы. Массивы могут содержать от 1 до 16 измерений. При необходимости размеры массива по каждому из измерений указываются в квадратных скобках. Например, VesMes[12], Abc[10,4,5].

Границы по измерению могут быть указаны явно, например VesT[5:8]. В этом случае массив будет состоять из четырех элементов: VesT[5], VesT[6], VesT[7], VesT[8].

Заметим, что массив не может быть элементом массива. *Нумерация элементов массива начинается с 1, если границы не были заданы явно.*

4.2. Домены

Столбцы в различных таблицах базы данных могут содержать однотипные данные. Кроме того, виды контроля этих данных также могут быть одинаковыми. В этих условиях описания данных и методов их контроля целесообразно выполнить один раз, тогда при описании таблиц достаточно указывать только имя соответствующего описания данных.

Для этих целей и служат описания доменов. (Напомним, что под доменами отношения R, где $R \subset A_1 \times A_2 \times \ldots \times A_n$, понимаются множества A_i $i \in [1, n]$.)

Перед тем как создавать столбцы, которые ссылаются на домены, необходимо задать описания доменов. Для этих целей существует команда CREATE DOMAIN. В результате ее выполнения создается шаблон, на который можно ссылаться в командах создания и модификации таблиц (CREATE TABLE и ALTER TABLE - см. раздел 4.3).

Наиболее полезно использование доменов, когда большое число таблиц содержит идентичные типы данных.

Столбцы, базирующиеся на доменах, наследуют все характеристики домена, причем часть из них может быть переопределена в локальных описаниях столбцов.

Поскольку описания доменов не связаны ни с какими таблицами, то ограничения логической целостности таблиц, естественно, не могут фигурировать в описании доменов.

Прежде чем создавать домены, необходимо выполнить проектирование базы данных, включающее проектирование системы таблиц и их нормализацию.

Далее следует создать, если это не было сделано ранее, саму базу и выполнить команду соединения с базой (см. CREATE DATABASE ... и CONNECT <database> USER <username> PASSWORD <password>).

Рассмотрим синтаксис описания доменов.

Создание доменов

```
CREATE DOMAIN domain [AS] <datatype>
[DEFAULT { literal \ NULL \ USER}})
[NOT NULL] [CHECK ( <dom_condition>)][COLLATE collation];

domain - имя создаваемого домена.
datatype - любой допустимый в InterBase тип данных.
< da ta type> ::= {
{SMALLINT | INTEGER \ FLOAT | DOUBLE PRECISION}[ <ar-
ray_dim>]
| {DECIMAL | NUMERIC} [(precision [, scale])] [<array_dim>]
|{DATE | TIME | TIMESTAMP) [<array_dim>]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
[( int)] [<array_dim>] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER | NATIONAL CHAR}
[VARYING] [(int)][<array_dim>]
| BLOB [SUB_ {int | subtype_name}][SEGMENT SIZE int]
[CHARACTER SET charname]
| BLOB [( seglen [, subtype])]
}
<array_dim> ::= [LIST_<dims>]
<dims> ::= x[::y]
```

Замечание 1. Квадратные скобки при описании массива являются *синтаксическим элементом, а не признаком его необязательности.*

Замечание 2. Тип данных не может быть переопределен при использовании домена в описании таблиц.

Замечание 3. Типы данных **TIME** и **TIMESTAMP** допустимы в версиях, начиная с 6.0..

Подробнее о типах данных уже говорилось выше.

DEFAULT {literal | *NULL* | USER} - задание значения по умолчанию. Значения по умолчанию присваивается соответствующему атрибуту при создании новой строки в таблице, если его значение не указано явно (например, отсутствует в списке ввода команды Insert). literal указывает значение явно, NULL задает признак "Нет значения", USER - имя пользователя, создающего запись. Для полей типа "дата" можно указать NOW - в этом случае вводится текущая дата.

CHECK - dom condition задает ограничение (описание контроля данных при вводе и изменении).

```
<dom_condition> : = {
VALUE <operator> <val>
/ VALUE [NOT] BETWEEN <val> AND <val>
/ VALUE [NOT] LIKE <val> [ESCAPE <val>]
/ VALUE [NOT] IN ( LIST_<val>)
```

```
| VALUE IS [NOT] NULL
| VALUE [NOT] CONTAINING <val>
| VALUE [NOT] STARTING [WITH] <val>
| ( <dom_condition>)
| NOT <dom_condition>
| <dom_condition> OR <dom_condition>
| <dom_condition> AND <dom_condition>
}
<operator> ::= { = | < | > | <= | >= | !< | !> | <> | != }
```

Подчеркнем, что при использовании конструкции CHECK необходимо помнить следующее:

- Домены создаются независимо друг от друга и, тем более, от каких-либо таблиц, следовательно, CHECK в домене не может ссылаться ни на какой другой домен или столбец таблицы.
- Домен может иметь только одну конструкцию CHECK.
- Конструкция CHECK не может быть переопределена при описании атрибута (поля) таблицы. Если при описании поля в таблице, ссылающегося на домен, имеющий CHECK, указана своя конструкция CHECK, то *действовать будут оба ограничения,* то есть будет работать конструкция CHECK (<dom_condition>) AND (<table_condition>).

Пример 4.1

CREATE DOMAIN MVEIGHT AS DOUBLE PRECISION [1:12];

Объявляет тип массива из 12 элементов.

Пример 4.2

CREATE DOMAIN USERNAME AS VARCHAR(20) DEFAULT USER;

Объявляет тип поля для хранения имени пользователя. Если имя пользователя при создании новой записи явно не указывается, то в него будет заноситься имя того пользователя, который создает запись.

Пример 4.3

CREATE DOMAIN MONTH AS SMALLINT CHECK(VALUE BETWEEN 1 AND 12);

Объявляет тип поля для хранения номера месяца с указанием ограничения для контроля вводимых данных.

Пример 4.4

```
CREATE DOMAIN D_ELEM AS CHAR(2) CHECK (VALUE IN ('Au', 'Ag',
'Pt', 'Pd', 'Os', 'Ir', 'Rb', 'Rt'));
```

Задает допустимые значения для типа полей задания драгоценных металлов.

Пример 4.5

```
CREATE DOMAIN PVEIGHT AS NUMERIC(12, 2) DEFAULT NULL CHECK
((VALUE IS NULL) OR (VALUE>1.25));
```

Задает допустимые значения для типа полей задания весовых характеристик (если вес не задан, то NULL, иначе - не меньше технологически минимального веса для данного производства).

Теперь при создании таблицы можно использовать введенные домены.

Пример 4.6

```
CREATE TABLE ABX (
 A MVEIGHT,
 B USERNAME,
 C MONTH,
 D D_ELEM,
 E PVEIGHT,
 KEY INTEGER,...
);
```

Изменение доменов

Изменение доменов осуществляется командой ALTER DOMAIN. С помощью команды ALTER DOMAIN можно изменить любые характеристики домена, *кроме типа данных (включая размеры массива) и установок NOT NULL.* Сделанные изменения воздействуют на атрибуты (поля) всех таблиц, где использовался измененный домен.

Команду ALTER DOMAIN может выдать либо его создатель, либо пользователь SYSDBA, либо другой пользователь с правами системного администратора.

Для изменения типа поля или установки NOT NULL необходимо удалить домен командой DROP DOMAIN, если это возможно (если домен используется для описания столбцов каких-либо таблиц, то удалить его нельзя, поскольку они при этом «подвисают»), а затем создать его снова с требуемыми характеристиками.

Синтаксис команды ALTER DOMAIN выглядит следующим образом:

```
ALTER DOMAIN name {
 [SET DEFAULT { literal / NULL / USER}]
```

Описание данных на основе SQL 85

```
] /DROP DEFAULT/
| /ADD CONSTRAINT CHECK ( <dom_condition>) ]
};/DROP CONSTRAINT/
};
```

Конструкции <dom_condition>, name и literal имеют тот же синтаксис, что и в команде CREATE DOMAIN.

Приведем примеры использования команды ALTER DOMAIN, основываясь на примерах для CREATE DOMAIN.

Пример 4.7

```
ALTER DOMAIN D_ELEM DROP CONSTRAINT;
ALTER DOMAIN D_ELEM ADD CHECK( value in ('H', 'Li',
'Na',
K'));
```

Выполнить замену ограничения одной командой нельзя! По синтаксису команды ограничение можно или удалить или добавить, прямая замена невозможна, так что замену следует делать в два этапа.

Пример 4.8

```
ALTER DOMAIN USERNAME SET DEFAULT '***';
```

Замена значения по умолчанию. Новая установка заменяет старую.

Удаление доменов

Удаление доменов осуществляется командой DROP DOMAIN. С помощью этой команды можно удалить описание домена.

Если домен используется в каких-либо таблицах, то удалить его нельзя (соответствующая команда завершится аварийно). В такой ситуации необходимо предварительно удалить в таблицах все столбцы, использующие домен, который мы хотим удалить.

Синтаксис команды DROP DOMAIN выглядит следующим образом:

DROP DOMAIN name;

Пример 4.9

```
DROP DOMAIN D_ELEM;
```

4.3. Таблицы

Перед тем как перейти к созданию таблиц, необходимо выполнить проектирование базы данных и нормализацию таблиц.

Далее следует определить, какие таблицы и с какими столбцами (полями) подлежат созданию.

Создать, если это не было сделано ранее, саму базу и выполнить команду соединения с базой (см. CREATE DATABASE ... и CONNECT <database> USER <username> PASSWORD <password>).

Создать комплект необходимых доменов и только после этого можно перейти к физическому вводу описаний таблиц.

Создание таблицы, точнее, ее описания и "пустографки" осуществляется командой CREATE TABLE.

При создании таблицы мы должны задать, как минимум, ее имя и перечень полей с их атрибутами и контрольными ограничениями. Кроме того, при создании таблицы можно задать ее первичный ключ, внешние ключи, задающие требования по поддержанию логической целостности, дополнительные виды контроля на уровне записей.

Создание таблиц. Команда CREATE TABLE

Команда CREATE TABLE имеет следующий синтаксис:

```
CREATE TABLE table [EXTERNAL [FILE] " <filespec>"]
 (LIST_<col_def>[, LIST_<tconstraint>]);
```

table - имя создаваемой таблицы. Имя таблицы внутри базы должно быть уникальным,

EXTERNAL [FILE] "<filespec>" задает таблицу, данные которой размещаются во внешней (не InterBase) таблице или файле,

<col_def> - описание поля (атрибута в терминах отношений) таблицы,

<tconstraint> - описание ограничений логической целостности для таблицы в целом.

Описание полей таблицы

Для описания полей <col_def> используется следующий синтаксис:

```
<col_def> ::= col {datatype | COMPUTED [BY] (< expr>) / domain}
[DEFAULT {literal | NULL | USER}]
[NOT NOLL] [<col_constraint>]
[COLLATE collation]
```

Первая строка относится к обязательным атрибутам описания столбца (поля):

col - имя столбца; должно быть уникальным в пределах таблицы;

{ datatype \ COMPUTED [BY] (< expr>) \ domain) задают тип данных в столбце, где *datatype* - любой допустимый в InterBase тип данных, а именно:

```
{SMALLINT | INTEGER | FLOAT | DOUBLE PRECISION} [ <array_dim>]
| {DECIMAL | NUMERIC} [(precision [, scale])] [ <array_dim>]
| {DATE | TIME | TIMESTAMP} [<array_dim>]
| {CHAR | CHARACTER | CHARACTER VARYING | VARCHAR}
[( int)] [<array_dim>] [CHARACTER SET charname]
| {NCHAR | NATIONAL CHARACTER \ NATIONAL CHAR}
[VARYING] [(int)][<array_dim>]
| BLOB [SUB_TYPE {int | subtype_name}][SEGMENT SIZE int]
[CHARACTER SET charname]
| BLOB [( seglen [, subtype])]
}
<array_dim> ::= [LIST_<dims>]
<dims> ::= x[;y]
```

ТИПЫ данных ***TIME и TIMESTAMP*** *допустимы в версиях, начиная с 6.0.*

domain — имя определенного ранее домена (описание домена, в свою очередь, содержит описание типа данных);
COMPUTED [BY] (< expr>) - выражение для вычисляемого столбца. Значения вычисляемых столбцов рассчитываются в соответствии с задаваемым выражением всякий раз при обращении к ним. Выражение может быть любым допустимым в InterBase выражением, возвращающим единственное значение простого типа (не массив). Например, допустимы выражения с конструкцией SELECT. Используя нашу базу, можно добавить в таблицу TBOOK_AUTHOR два поля B1 и B2, которые будут использоваться для выбора автора и названия книги. Для добавления полей используется команда ALTER TABLE, синтаксис которой будет рассмотрен ниже.

Пример 4.10

```
alter TABLE TBOOK_AUTHOR add
 B1 varchar(60) COMPUTED BY ((select a.auname from tauthora
   where a.author=tbook_author.author));

alter TABLE TBOOK_AUTHOR add
 B2 COMPUTED BY ((select a.booknm from tbook a
   where a.unikey=tbook_author.bookkey));
```

Тогда результат работы

```
select * from tbook_author;
```

можно представить в виде таблицы.

Таблица 4.1. *Перечень книг с указанием авторов и названий (вычисляемые поля)*

Unikey	*Author*	*Bookkey*	*B1*	*B2*
58	25	6	Культин Н.Б.	Макрокоманды MS Word
59	33	12	Буассо Марк	Введение в технологию ATM
60	34	12	Деманж Мишель	Введение в технологию ATM
61	35	12	Мюнье Жан-Мари	Введение в технологию ATM
62	32	11	Луис Дерк	C и C++ Справочник
63	31	10	Дунаев Сергей	Borland-Технлогии. SQL-Link InterBase, Paradox for Windows, Delphi
64	29	9	Елманова Н.З.	Введение в C++ Builder
65	30	9	Кошель С П.	Введение в C++ Builder
66	28	8	Подбельский Вадим Валериевич	Язык C++
67	26	7	Хаселир Райнер Г.	Word 6 for Windows
68	27	7	Фаненштих Клаус	Word 6 for Windows
69	21	16	Ладыжинская Ольга Александровна	Математические вопросы динамики вязкой несжимаемой жидкости
70	24	15	без авторов	Тесты. Сборник 11 класс. Варианты и ответы государственного тестирования. Пособие для подготовки к тестированию
71	23	14	Розенталь Д.Э.	Справочник по правописанию и литературной правке
72	22	13	Бурова И.И.	The history of England. Absolute Monarchy

Unikey	*Author*	*Bookkey*	*B1*	*B2*
73	19	17	Дашкова Полина	Кровь нерожденных
74	20	18	Хмелевская Иоанна	Тайна

Вторая строка в синтаксисе описания полей (DEFAULT { literal | NULL | USER}) задает значение поля при создании новой строки таблицы.

При добавлении в таблицу новых строк не требуется, вообще говоря, задание значений всех ее столбцов. В то же время каждое поле в таблице должно иметь хоть какое-то допустимое значение. Для того чтобы задать значение столбца в этом случае, и служит конструкция DEFAULT. Если она отсутствует, то это эквивалентно конструкции DEFAULT NULL.

Если тип поля задан с помощью Домена, в котором присутствует конструкции DEFAULT, а в описании поля таблицы DEFAULT отсутствует, то в качестве значения по умолчанию принимается значение, заданное в описании домена. Если конструкция DEFAULT в описании поля таблицы указана явно, то значение по умолчанию будет взято из нее вне зависимости от указанного в описании домена (значение переопределяет заданное в домене).

literal задает значение поля явно и по типу должно соответствовать объявленному полю. В качестве допустимого значения для полей типа DATE может быть сегодняшняя дата, задаваемая конструкцией NOW. *NULL* задает значение NULL Поскольку оно предполагается изначально, то задавать его стоит только при необходимости переопределить значение, указанное в домене. Последнее возможно, если в домене не указана конструкция NOT NULL, которую уже нельзя отменить. *USER* указывает, что в поле будет занесено символьное имя USER'a, создавшего запись. Длина этого поля должна соответствовать требованиям к длине для имен пользователей (обычно от 8 до 16 символов) и использовать ту же или совместимую кодовую таблицу (character set).

Пример 4.11

```
CREATE DOMAIN USERNAME AS VARCHAR(20)
CREATE TABLE ABC (
 ABC_DATE DATE DEFAULT "NOW",
 ABC_USER USERNAME DEFAULT USER,
 ABC_COUNT DOUBLE PRECISSION DEFAULT 1
);
```

NOT NULL из третьей строки указывает, что поле не может содержать значение NULL (ни при создании, ни при обновлении данных). Отметим, что явно задаваемое значение NULL не должно конфликтовать

с конструкцией NOT NULL, как в COUNT INTEGER DEFAULT NULL NOT NULL.

<col_constraint> из этой же строки описывает ограничения логической целостности для столбца.

InterBase позволяет задавать ограничения на отдельные столбцы или таблицу в целом, называемые ограничениями логической целостности, которые задают порядок управления зависимостями между столбцом и таблицей или таблицей и таблицей. Они воздействуют на все выполняемые с таблицей действия, автоматически поддерживаясь системой. В зависимости от типа ограничения они применяются либо к таблице целиком, либо к отдельным ее столбцам.

Конструкция <col_constraint> относится к отдельному столбцу и имеет следующий синтаксис:

```
<col_constraint> ::= [CONSTRAINT constraint]
{UNIQUE
| PRIMARY KEY
| CHECK (<search_condition>)
| REFERENCES other_table [(LIST_other_col)]
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
}
```

Ограничения на первичный (PRIMARY KEY) или уникальный (UNIQUE) ключ означают, что значение в соответствующем столбце является уникальным, то есть в таблице не может быть двух строк с одинаковыми значениями в данном столбце. Соответственно значения в таком столбце не могут принимать значение NULL. При попытке записать значение, которое уже встречается в таблице, InterBase выдает сообщение об ошибке.

Отметим, что и первичные, и уникальные ключи могут строиться не только по отдельному полю, но и по группе полей (см. описание ограничений логической целостности для таблицы в целом).

В таблице может быть только один первичный ключ. Уникальных ключей может быть несколько, с точки зрения теории отношений все они являются синонимами первичного ключа. В то же время трудно представить себе ситуацию, когда может понадобиться иметь более одного уникального ключа, отличного от первичного.

Другие элементы конструкции <col_constraint> означают:

- *CHECK (<search_condition>)* задает условие, которому должно соответствовать значение определяемого столбца. Синтаксис конструкции CHECK следующий (см. также <search_condition> в команде SELECT):

Описание данных на основе SQL 91

```
CHECK (<search condition>);
<search_condition> ::= {<val> <operator>
{<val> / (<select_one>)}
} <val> [NOT] BETWEEN <val> AND <val>
/ <val> [NOT] LIKE <val> [ESCAPE <val>]
/ <val> [NOT 7 IN (LIST_<val> / <select_list>)
/ <val> IS [NOT] NULL
/ <val> {[NOT] {= / < / >} / >= / <=}
{ALL / SOME / ANY; (<select_list>)
/ EXISTS ( <select_expr>)
/ SINGULAR (<select_expr>)
/ <val> [NOT] CONTAINING <val>
/ <val> [NOT] STARTING [WITH] <val>
/ (<search_condition>)
/ NOT <search_condition>
/ <search_condition> OR <search_condition>
/ <search_condition> AND <search_condition>}
```

Конструкция CHECK относится только к одной строке таблицы. Для столбца может быть задана только одна конструкция CHECK.

Если описание столбца базируется на домене, то введенная конструкция CHECK не отменяет заданную в домене, а добавляет собственный контроль к заданному в домене.

Отметим, что в конструкции <search_condition> можно использовать любые данные текущей строки таблицы, а также результаты поиска по базе, что делает конструкцию CHECK значительно более мощной, чем ее аналог в описании домена.

- **REFERENCES** other_table *[(LIST_other_col)]*

/ON DELETE /NO ACTION|CASCADE|SET DEFAULT|SET NULL]]
[ON UPDATE /NO ACTION|CASCADE|SET DEFAULT|SET NULL]]

задает ограничение внешнего ключа для описываемого столбца.

Ограничение означает, что данное поле соответствует первичному ключу *[(LIST_other_col)]* в таблице *other_table* и в этой таблице имеется строка с указанным значением. Если список опущен, то предполагается список из одного поля, имеющего то же имя, что и описываемое.

Дополнительные режимы *ON DELETE* и *ON UPDATE* задают действия, производимые при удалении или обновлении ключевых полей в родительской (здесь *other_table*) таблице.

Возможны следующие варианты:

NO ACTION – нет действий (принимается по умолчанию).

CASCADE - каскадное удаление (замена) влечет удаление (замену) во всех строках дочерней таблицы при удалении, замене соответствующих им строк родительской таблицы.

SET DEFAULT - при удалении (замене) строк родительской таблицы, соответствующие им поля в дочерней переустанавливаются в значения по умолчанию.

SET NULL - при удалении (замене) строк родительской таблицы, соответствующие им поля в дочерней переустанавливаются в NULL.

Наконец четвертая строка синтаксиса описания полей (COLLATE collation) задает порядок сравнения символьных данных (для алфавитного упорядочения).

Нам осталось рассмотреть последний элемент синтаксиса команды CREATE TABLE, а именно ***<tconstraint>*** - описание ограничений логической целостности для таблицы в целом (см. также описание ограничений логической целостности <col_constraint> для столбца).

Приведем синтаксис конструкции <tconstraint>.

```
<tconstraint> : := /"CONSTRAINT constraint./
{{PRIMARY KEY | UNIQUE.? (LIST_col)
/ FOREIGN KEY (LIST_col) REFERENCES other_table
[ON DELETE {NO ACTION / CASCADE / SET DEFAULT / SET NULL}]
[ON UPDATE {NO ACTION / CASCADE / SET DEFAULT / SET NULL}]
/ CHECK ( <search_condition>)}
```

Ограничение *{{***PRIMARY KEY | UNIQUE***}* (*LIST*_col) на первичный (PRIMARY KEY) или уникальный (UNIQUE) ключ означают, что значение в указанном столбце или группе столбцов является уникальным, то есть в таблице не может быть двух строк с одинаковыми значениями в данном столбце или группе столбцов. Соответственно значения в таких столбцах не могут принимать значение NULL. При попытке записать значение, которое уже встречается в таблице, InterBase выдает сообщение об ошибке.

Ограничения **FOREIGN KEY** (*LIST*_col) **REFERENCES** other_table *[***ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL/7** на внешний (FOREIGN KEY) ключ означают, что значение в указанном столбце или списке столбцов *[(LIST*_col)*]* соответствует первичному ключу в таблице *other_table* и в этой таблице имеется строка с указанным значением. Соответственно значения в таких столбцах не могут принимать значение NULL.

Дополнительные режимы *ON DELETE* и *ON UPDATE* задают действия, производимые при удалении или обновлении ключевых полей в родительской (здесь *other_table*)таблице.

Возможны следующие варианты:

NO ACTION- нет действий (принимается по умолчанию).

Описание данных на основе SQL 93

CASCADE - каскадное удаление (замена) влечет удаление (замену) во всех строках дочерней таблицы при удалении, замене соответствующих им строк родительской таблицы.

SET DEFAULT — при удалении (замене) строк родительской таблицы соответствующие им поля в дочерней таблице переустанавливаются в значение по умолчанию.

SET NULL - при удалении (замене) строк родительской таблицы соответствующие им поля в дочерней таблице переустанавливаются в NULL.

CHECK (<search_condition>) задают условия, проверяемые по значениям группы столбцов. Синтаксис конструкции *<search_condition>* аналогичен соответствующей конструкции команды SELECT.

Замечание. Конструкция **FOREIGN KEY** (*LIST*_col) **REFERENCES** other_table ... для таблицы, как и конструкция **REFERENCES** other_table *[(LIST*_other_col)*]* для отдельного столбца порождают внешний ключ, но при задании списка столбцов в первом случае перечисляются столбцы *описываемой таблицы,* а во втором той *таблицы, на которую осуществляется ссылка.*

Использование внешних файлов

Конструкция использования внешних файлов EXTERNAL FILE имеет вид

```
EXTERNAL [FILE] "<filespec>"
```

С помощью этой конструкции создается описание таблицы. Сами данные размещаются во внешнем файле (таблице), отличном от интербейсовской базы. Внешние файлы представляют собой тексты, которые могут обрабатываться неинтербейсовскими приложениями. Согласно синтаксису для команды CREATE TABLE, спецификация файла, следующая за словом EXTERNAL, представляет полностью специфицированное имя файла (включая путь). Файл может модифицироваться вне InterBase, так как доступ к нему осуществляется только по мере необходимости.

Конструкция EXTERNAL FILE используется для:

- Импорта данных из внешнего файла жесткого формата (с записями фиксированной длины) в новую или существующую таблицу InterBase. Это позволяет обновлять таблицы InterBase данными из внешних источников. Многие приложения позволяют создавать подобные внешние файлы с записями фиксированной длины.
- Выборки данных с помощью SELECT из внешних файлов так же, как если бы они были стандартными таблицами InterBase.

- Экспорта данных из существующих таблиц InterBase во внешний файл. Для форматирования данных из внутренних таблиц InterBase во внешний файл с записями фиксированной длины для использования другими приложениями.

На использование внешних файлов накладываются следующие ограничения:

- Прежде чем получить доступ к внешним файлам из базы данных, они должны быть созданы.
- Все записи во внешнем файле должны быть одной длины. Во внешнем файле нельзя использовать такие типы данных, как массив или BLOB.
- Когда создается таблица, которая будет использоваться для импорта внешних данных, в ней должно быть определено поле для хранения символов конца строки (типа перевод каретки, возврат строки). Размер поля должен быть достаточен для хранения таких символов (обычно один или два байта). В большинстве версий Unix - 1 байт. Для Windows, NT и NetWare - 2 байта.
- Несмотря на то, что допустимо чтение числовых данных прямо из внешней таблицы, значительно проще читать их как символьные данные и преобразовывать с помощью функции CAST.
- Данные, трактуемые как VARCHAR в InterBase, должны храниться во внешнем файле в следующем формате: <2-byte unsigned shortxstring of character bytes>, где *2-byte unsigned short* содержит длину в байтах строки, непосредственно следующей за ней. Поскольку таким образом нельзя обеспечить должной совместимости, использование данных типа VARCHAR во внешних файлах не рекомендуется.

При работе с внешними таблицами разрешены только команды INSERT для добавления в них данных и SELECT для выборки. Команды UPDATE и DELETE использовать нельзя. При попытке их применения InterBase выдаст сообщение об ошибке.

Вставка данных во внешние таблицы и выборка из них не используют стандартного механизма транзакций, поскольку они находятся вне базы данных, поэтому изменения выполняются сразу и механизм отката для них невозможен. Для использования механизма управления транзакциями необходимо создать внутреннюю InterBase таблицу и перенести в нее данные из внешней таблицы.

Если удаляется база (DROP DATABASE) необходимо также удалить и внешние файлы, автоматически они удаляться не будут.

Импорт внешних файлов в таблицы InterBase

Прежде чем начать работу с внешним файлом, необходимо создать его описание, используя команду CREATE TABLE <file> EXTERNAL FILE "<extern_file_name>". Поля во внешнем файле следует описать как символьные строки фиксированной длины (CHAR). Последнее поле должно быть зарезервировано для символов конца строки (для Unix - 1 байт, для Windows - 2 байта). Сам файл должен быть расположен на сервере.

Рассмотрим пример создания внешней таблицы.

В Excel создана таблица с тремя столбцами, содержащими соответственно фамилию, год рождения и табельный номер. Таблица Excel сохранена как форматированный текст (с пробелами в качестве разделителей) List.prn.

Замечание. Чтобы не создавать себе лишних проблем, сначала следует создать такой внешний файл, выяснить его формат и только потом создавать соответствующее описание в InterBase. В противном случае придется либо менять формат созданного файла, что может быть очень неудобно для внешних приложений, либо переописывать соответствующую внешнюю таблицу InterBase.

Создано описание внешней таблицы E_LIST в InterBase.

Пример 4.12

```
create table E_LIST EXTERNAL "d:\LINK\LISPRN"(
 FAMILIA CHAR(12) NOT NULL,
 GOD_ROG  CHAR(4),
 TAB_NOM  CHAR(8),
 RAZFDEL CHAR(2)
);
```

Теперь запрос

```
select Familia, god_rog, Tab_nom from E_LIST;
```

дает следующий результат:

Таблица 4.2. *Список во внешнем файле, экспортированном из Excel*

FAMILIA	GOD_ROG	TAB_NOM
Сидоров	1916	1
Иванов	1989	1003
Петров	1958	2087
Мухин	1975	1312

Если данные из внешней таблицы предполагается каким-либо образом обрабатывать, особенно, если обработка предполагает внесение изменений, то ее хранение в такой форме, безусловно, неудобно. В этом случае целесообразно создать ее аналог в формате InterBase, для которой можно ввести необходимые индексы, триггеры и другие средства эффективной обработки. Столбец - аналог столбцу RAZFDEL - целесообразно сохранить, если в дальнейшем предполагается производить выгрузку во внешнюю таблицу, иначе он, вообще говоря, не нужен. Я бы рекомендовал сохранять его в любом случае. Никогда заранее* не знаешь, что будет в дальнейшем, а затраты на его хранение невелики.

Продолжим наш пример. Для этого создадим обычную таблицу InterBase.

Пример 4.13

```
create table I_LIST(
 FAMILIA CHAR (12) NOT NULL,
 GOD_ROG integer,
 TAB_NOM integer,
 RAZFDELCHAR(2),
 primary key(FAMILIA)
);
```

Теперь заполним ее данными из внешней таблицы.

```
insert into I_LIST (FAMILIA, GOD_ROG, TAB_NOM, RAZFDEL)
 select FAMILIA, CAST(GOD_ROG as integer),
  CAST(TAB_NOM as integer),
  RAZFDEL from e_list;
```

Затем проведем необходимую обработку и, наконец, снова выгрузим данные во внешний файл. Предварительно его следует очистить (например, текстовым редактором), иначе новые данные просто припишутся к старым, а это вряд ли то, что хотелось бы получить.

```
insert into E_LIST (FAMILIA, GOD_ROG, TAB_NOM, RAZFDEL)
 select FAMILIA, CAST(GOD_ROG as CHAR(4)),
  CAST(TAB_NOM as CHAR(8)),
  RAZFDEL from i_list;
```

Следует отметить, что из-за преобразований внешний вид текста во внешнем файле может измениться. Прежде всего, это касается выравнивания текста и положения десятичной точки.

Замечание 1. При вводе данных из внешней таблицы следует обратить внимание на синтаксическую правильность ее заполнения. Это касается прежде всего:

Описание данных на основе SQL 97

- контроля длины всех строк и отдельных полей;
- правильности заполнения разделителей строк;
- если таблица содержит уникальные ключи, то контроль этого должен быть обязательно проведен во внешнем приложении, иначе оператор Insert выдаст сообщение об ошибке.

Замечание 2. При разработке приложений в среде специализированных средств программирования, таких как C++Builder или Delphi, проблемы с внешними таблицами обычно вообще не возникают. В этом случае для выборки данных из внешних файлов можно написать запрос средствами локального SQL, а затем, используя компоненту типа BatchMove, перенести данные в таблицу InterBase или наоборот. Так что, если вы работаете с системами программирования подобного типа, то внешние таблицы вам, скорее всего, никогда не понадобятся и вы просто зря потеряли время на прочтение последней страницы, хотя, как знать, может быть, все-таки пригодится.

Модификация таблиц. Команда ALTER TABLE

Команда ALTER TABLE предназначена для изменения структуры уже существующих таблиц.

Она позволяет:

- добавить в таблицу новый столбец;
- удалить из таблицы существующий столбец;
- добавить в таблицу ограничение на столбец или таблицу;
- удалить из таблицы ограничение на столбец или таблицу.

В одной команде ALTER TABLE можно задать любое число изменений. Внесение изменений разрешено ее создателю, а также пользователям, имеющим права системного администратора (SYSDBA или другого пользователя с аналогичными правами).

Модификация таблицы, особенно если она содержит данные и связана с другими таблицами, является потенциально опасной операцией. Поэтому прежде чем приступить к изменению таблицы, необходимо продумать схему внесения изменений.

Добавление столбца в таблицу

При добавлении нового столбца в таблицу каких-либо конфликтов с ранее введенными данными не возникает. Тем не менее, если столбец имеет ограничения, в том числе NOT NULL, и таблица уже содержит данные, то необходимо позаботиться, чтобы после создания столбца записанные в него данные имели допустимое значение. Для этого можно, например, указать значение по умолчанию (DEFAULT).

Удаление столбца из таблицы

Из таблицы можно удалить столбец только в том случае, если он не используется ни в каких ограничениях. Таковыми могут быть CHECK-конструкции по таблицам, индексы, а также триггеры и хранимые процедуры. Поэтому прежде чем удалить столбец необходимо удалить все ограничения, в которых этот столбец фигурирует, в том числе и относящиеся совсем к другим таблицам (например, внешние ключи или триггеры).

Изменение формата столбца

Стандартный синтаксис команды ALTER TABLE вообще не предполагает возможности изменения столбца. Поэтому изменение предполагает, по крайней мере, две команды ALTER TABLE. Первая - удалить столбец и вторая - добавить столбец. При выполнении этих операций следует руководствоваться замечаниями по добавлению и удалению. Следует, конечно, помнить, что удаление столбца влечет полную потерю хранящихся в нем данных. Если эти данные необходимо сохранить, то операция замены распадается на следующие:

- добавить новый столбец с тем же типом данных, что и в изменяемом столбце (ALTER TABLE ADD …);
- скопировать данные из существующего столбца в новый (UPDATE...);
- выполнить необходимые действия по подготовке к удалению существующего столбца (удаление соответствующих ограничений);
- выполнить удаление существующего столбца (ALTER TABLE DROP…);
- создать столбец с именем удаленного и новыми характеристиками (ALTER TABLE ADD …);
- скопировать данные из столбца-копии в столбец с измененными характеристиками (UPDATE …=CAST(…)…);
- удалить столбец-копию (ALTER TABLE DROP …);
- выполнить операции по восстановлению (при необходимости) группы ограничений, удаленных в процессе подготовки к удалению исходного столбца;
- воскликнуть ура, все получилось!

Добавление ограничений

Добавление ограничений не требует никаких вспомогательных действий. Необходимо только в случае добавления нескольких ограничений соблюдать порядок их добавления.

Удаление ограничений

Удаление ограничений также не требует никаких вспомогательных действий, правда и здесь в случае удаления нескольких ограничений следует соблюдать порядок их удаления.

Команда ALTER TABLE имеет следующий синтаксис:

```
ALTER TABLE table LIST_<operation>;

<operation> ::= {ADD <col_def>
| ADD <tconstraint>
| DROP col
| DROP CONSTRAINT constraint.?

<col_def> = col {<datatype> / COMPUTED [BY] {< expr>) /
domain}
/DEFAULT {literal / NULL / USER}]
[NOT NULL]
[<col_constraint>]
/•COLLATE collation./

< datatype> ::= {
SMALLINT / INTEGER / FLOAT / DOUBLE PRECISION;
[<array_dim>]
| {DECIMAL / NUMERIC; [{ precision [, scale7 ))
[<array_dim>]
/ {DATE / TIME / TIMESTAMP} [ <array_dim>]
/ (CHAR /CHARACTER / CHARACTER VARYING / VARCHAR}
[( int) J [<array_dim>] [CHARACTER SET charname]
/ {NCHAR / NATIONAL CHARACTER / NATIONAL CHAR}
[VARYING] [{ int); [ <array_dim>]
/ BLOB [SUB_TYPE { int / subtype_name}] /'SEGMENT SIZE int;
[CHARACTER SET charname]
/ BLOB [( seglen [, subtype;);
)

<array_dim> ::= [LIST_<dims>]
<dims> ::= [x:]y

< expr> = Любое допустимое в SQL выражение, дающее в результате единственное значение.

<col_constraint> ::= [CONSTRAINT constraint] <constraint_def>

<constraint_def> ::= {UNIQUE / PRIMARY KEY
/ REFERENCES other_table [(LIST_other_col)]}
[ON DELETE {NO ACTION / CASCADE / SET DEFAULT / SET NULL}]
[ON UPDATE {NO ACTION / CASCADE / SET DEFAULT / SET NULL}]
| CHECK ( <search_condition>)
```

```
<tconstraint> ::= [CONSTRAINT constraint]
{{PRIMARY KEY / UNIQUE} (LIST_col)
| FOREIGN KEY (LIST_col) REFERENCES other_table
[ON DELETE (NO ACTION / CASCADE / SET DEFAULT / SET NULL}]
[ON UPDATE {NO ACTION / CASCADE / SET DEFAULT / SET NULL}]
/ CHECK ( <search_condition>)}

<search_condition> =
{ <val> <operator> { <val> / ( <select_one>)}
| <val> [NOT] BETWEEN <val> AND <val>
/ <val> [NOT] LIKE <val> /'ESCAPE <val>]
/ <val> [NOT] IN ( <val> [ , <val> ...] / <select_list>)
/ <val> IS /HOT; NULL
/ <val> {>= | <=} | [NOT] {= | < | >}
{ALL / SOME / ANY} (<select_list>)
/ EXISTS ( <select_expr>)
/ SINGULAR ( <select_expr>)
/ <val> [NOT] CONTAINING <val>
/ <val> [NOT] STARTING [WITH] <val>
/ ( <search_condition>)
/ NOT <search_condition>
/ <search_condition> OR <search_condition>
/ <search_condition> AND <search_condition>}

<val> = {
col [<array_dim>] / : variable
/ <constant> / <expr> / <function>
/ udf {[ <val> [, <val> ...]])
| NULL / USER / RDB$DB_KEY / ?

[COLLATE collation]
<constant> = num / ' string' / charsetname ' string'

<function> = {
COUNT (* / [ALL] <val> / DISTINCT <val>)
/ SUM ([ALL] <val> / DISTINCT <val>)
/ AVG ([ALL] <val> / DISTINCT <val>)
/ MAX ([ALL] <val> / DISTINCT <val>)
/ MIN ([ALL] <val> / DISTINCT <val>)
/ CAST ( <val> AS <datatype>)
/ UPPER ( <val>)
/ GEN_ID ( generator, <val>)
}.ALTER TABLE

<operator> = {= /< /> /<= />= / !< / !> /<> / '•=}
```

<select_one> = SELECT, возвращающий одну строку с одним столбцом.

<select_list> = SELECT, возвращающий несколько (возможно, 0) строк с одним столбцом.
<select_expr> = SELECT, возвращающий несколько (возможно, 0) строк с несколькими столбцами.

Из описания синтаксиса видно, что элементы синтаксиса операции ADD полностью соответствуют аналогичным элементам описания столбцов или ограничений при создании таблиц (CREATE TABLE). В результате ее выполнения создается соответствующий элемент описания таблицы. Операция ADD может завершиться *безуспешно,* если:

- пользователь, выдавший команду, не имеет соответствующих прав;
- делается попытка добавить первичный или уникальный ключ в случае, если уже хранящиеся в таблице данные не соответствуют вводимому ограничению.

Операция DROP обеспечивает удаление указанного в ней элемента. Ее синтаксис достаточно прост и не нуждается в пояснении. В то же время необходимо отметить, что операция DROP может завершиться *безуспешно,* если:

- пользователь, выдавший команду, не имеет соответствующих прав;
- делается попытка удалить столбец, входящий в первичный, уникальный или внешний ключ;
- делается попытка удалить столбец, входящий в ограничения CHECK;
- делается попытка удалить столбец, используемый в вычисляемом столбце или триггере;
- делается попытка удалить столбец, входящий в ограничения CHECK, вычисляемый столбец или триггер других таблиц.

Таким образом, прежде чем удалить сам столбец из таблицы, необходимо удалить все ссылки на него.

Добавление столбцов

Пример добавления столбцов уже приводился, когда говорилось о вычисляемых столбцах (см. пример 4.10). Рассмотрим этот же пример, объединив обе команды в одну.

Пример 4.14

```
alter TABLE TBOOK_AUTHOR
 add B1 varchar(60) COMPUTED BY ((select a.auname from tau-
thor a
   where          a.author=tbook_author.author)),
 add B2 COMPUTED BY ((select a.booknm from tbook a
   where          a.unikey=tbook_author.bookkey));
```

Добавление ограничений

Приведем пример.

Пример 4.15

```
ALTER TABLE TBOOK_AUTHOR
 ADD CONSTRAINT UK_BOOK_AUTHOR UNIQUE (AUTHOR, BOOKKEY);
```

Удаление столбцов

Рассмотрим удаление только что добавленных столбцов.

Пример 4.16

```
alter TABLE TBOOK_AUTHOR    drop Bl, drop B2;
```

При удалении столбцов необходимо помнить, что нельзя удалять столбцы, входящие в ключи или используемые в триггерах, поскольку это нарушает целостность базы. Если это все же необходимо, то следует вначале удалить соответствующие ключи и триггеры.

Удаление ограничений

Приведем пример.

Пример 4.17

```
ALTER TABLE TBOOK_AUTHOR drop CONSTRAINT UK_BOOK_AUTHOR;
```

Удаление таблиц. Команда DROP TABLE

Команда DPOP TABLE предназначена для удаления таблиц. Ее синтаксис:

DROP TABLE name;

name - имя удаляемой таблицы.

Описание данных на основе SQL 103

В результате выполнения команды DPOP TABLE удаляется содержимое таблицы, ее описание, все связанные с таблицей индексы и триггеры.

Если в базе есть ссылки на удаляемую таблицу из других таблиц, представлений, ограничений логической целостности, то удаление не выполняется. Кроме того, удаление также не будет выполнено, если имеется активная транзакция, работающая с удаляемой таблицей.

Таким образом, перед удалением таблицы необходимо:

- завершить все транзакции, работающие с таблицей;

удалить все ссылки на удаляемую таблицу.

Поскольку не хотелось бы рушить нашу учебную базу, в **качестве** примера возьмем удаление несуществующей таблицы.

Пример 4.18

```
DROP TABLE ABC;
```

4.4. Индексы

Индексы предназначены для ускорения поиска данных на запросы в соответствии с заданными условиями. Кроме того, индексы могут использоваться для контроля уникальности ключевых выражений. В определенном смысле индекс является аналогом оглавления или алфавитного указателя (дающих номера страниц), то есть по значению индекса мы сразу находим место, где находятся интересующие строки таблиц.

В индексе хранятся значения индексированного столбца или столбцов наряду с указателями на все дисковые блоки, которые содержат строки с соответствующими значениями.

При выполнении запроса InterBase 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 InterBase 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, 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.

Индекс может быть определен как на отдельном столбце, так и на множестве столбцов таблицы.

Многоколоночные индексы могут использоваться для поиска с одиночным столбцом, если столбец, по которому ведется поиск, первый в индексе.

Тем не менее, индексирование оправдано 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. Их можно перестроить, но это также требует затрат времени.

В этих условиях при проектировании базы данных необходимо находить разумный компромисс между требованиями по ускорению поиска данных и требованию по скорости их обновления. В частности, использование индексов для небольших по объему таблиц вообще не оправдано. Если имеется индекс по группе полей, то поиск по первому из полей группы может прямо использовать этот индекс, следовательно, нет смысла по нему делать отдельный индекс. Если поиск по каким-либо полям редок, то построение по ним индекса неэффективно. Если данные в столбце могут принимать лишь несколько значений, то выделение группы данных с заданным значением ключа может сократить объем простого просмотра лишь в несколько раз. Это может и не компенсировать дополнительных затрат по поиску в индексе и большему времени выборки самих данных из-за выборки данных в порядке, отличном от естественного.

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.

К сожалению, более конкретные рекомендации по определению состава индексов едва ли возможны. Оптимальный выбор зависит и от структуры базы, и от характера ее использования.

Создание индексов (команда CREATE INDEX)

Индексы создаются либо пользователем с помощью команды CREATE INDEX, либо автоматически при выполнении команды CREATE TABLE. InterBase позволяет пользователям создавать до 64 индексов к таблице в версии 5 и до 2^{16} в версии 6. Чтобы создавать индексы, необходимы права на соединение с базой данных.

Отметим, что для просмотра всех индексов, определенных для текущей базы данных, следует использовать isql-команду SHOW INDEX. Для просмотра всех индексов, определенных для отдельной таблицы, исполь-

Описание данных на основе SQL 105

зуется команда SHOW INDEX tablename. Для просмотра конкретного индекса используется SHOW INDEX indexname.

InterBase автоматически генерирует индексы системного уровня по столбцу или набору столбцов, когда таблицы определяются с конструкциями ограничения PRIMARY KEY, FOREIGN KEY или UNIQUE.

Команда CREATE INDEX создает индекс на одном или нескольких столбцах таблицы. Одностолбцовый индекс используется для поиска по одному, многостолбцовый - по нескольким столбцам одновременно.

Опции команды определяют:

- порядок сортировки для индекса;

допустимость повторяющихся значений в индексированном столбце.

CREATE INDEX используется для:

- ускорения доступа к данным;

ускорения сортировки данных (см. опцию CREATE INDEX ORDER BY команды SELECT).

В процессе добавления и модификации данных индекс может стать несбалансированным, что приводит к замедлению при работе с ним. Для восстановления индекса следует использовать команды SET STATISTICS, либо деактивировать и реактивировать индекс командой ALTER INDEX.

Синтаксис команды CREATE INDEX:

```
CREATE [UNIQUE] [ASC [ENDING] / DESC[ENDING]]
INDEX index ON table (LIST_col);
```

ASC[ENDING] или DESC[ENDING] задает способ упорядочения данных.

ASC или ASCENDING - индекс создается по возрастанию ключей.

DESC или DESCENDING - индекс создается по убыванию ключей.

index задает имя индекса.

table задает имя индексируемой таблицы.

col задает имя столбца, по значениям которого строится индекс; таких столбцов может быть несколько.

UNIQUE задает режим уникального индекса. При задании этого режима блокируется запись в таблицу строк с одними и теми же значениями в столбцах, образующих индекс. Отметим, что первичный ключ таблицы также является и уникальным. В терминах теории отношений уникальный ключ является потенциальным ключом отношения и так же, как и первичный, может использоваться для идентификации строк в таблице.

В качестве примера рассмотрим команду создания уникального индекса по номерам читательских билетов в нашей базе. Вообще говоря, можно было бы создать уникальный индекс и по фамилиям читателей, но

это не вполне корректно, поскольку нельзя гарантировать, что в будущем не появятся полные однофамильцы.

Пример 4.19

```
CREATE  UNIQUE  ASCENDING
INDEX  TREADER_RDNUMB  ON  TREADER  (RDNUMB);
```

Замечание. При создании нового уникального индекса по уже существующей таблице эта операция завершится успешно только в том случае, если в заданном наборе столбцов действительно нет повторяющихся значений. Для проверки этого можно предварительно использовать команду SELECT. Применительно к индексу, созданному в приведенном выше примере, это может выглядеть так.

Пример 4.20

```
SELECT  RDNUMB,  COUNT(*) FROM  TREADER
GROUP BY RDNUMB
HAVING     COUNT(*)>1;
```

Если дублей нет, то в результате не будет ни одной строки, иначе получится список дублированных значений ключа.

Если предполагается упорядочение данных как по возрастанию, так и по убыванию ключа, то можно использовать соответственно два индекса. Например, к созданному индексу TREADER_RDNUMB можно добавить индекс по убыванию, хотя и не стоит объявлять его уникальным, чтобы не порождать лишнюю (во всех смыслах) проверку при вводе данных.

Пример 4.21

```
CREATE DESCENDING
INDEX  TREADER_RDNUMB_DS  ON  TREADER  (RDNUMB);
```

Как уже отмечалось, индекс может использоваться для **ускорения** сортировки данных. Последнее требование можно сформулировать **в** запросе с помощью опции PLAN явно.

Пример 4.22

```
SELECT  *  FROM  TREADER
  PLAN  (TREADER  ORDER  TREADER_RDNUMB)
ORDER  BY  RDNUMB
```

Стоит, правда, заметить, что даже, если не **указывать план явно,** InterBase выберет его в нашем случае самостоятельно.

Описание данных на основе SQL 107

КомандаALTERINDEX

Синтаксис:

```
ALTER INDEX name {ACTIVE / INACTIVE};
```

Команда ALTER INDEX, как видно из ее синтаксиса, не предназначена для явного изменения индекса. Она позволяет лишь деактивировать или, наоборот, повторно активировать неактивный индекс. При реактивации индекса осуществляется его перестроение. Последнее может быть использовано для оптимизации индекса, поскольку после большого количества изменений он может стать несбалансированным и, соответственно, время поиска по нему начнет расти.

Пример 4.22

```
ALTER INDEX TREADER_RDNUMB  INACTIVE;
ALTER INDEX TREADER_RDNUMB  ACTIVE;
```

На использование команды ALTER INDEX накладываются некоторые ограничения.

- Нельзя выполнить команду ALTER INDEX, если изменяемый индекс используется в данный момент, например командами изменения или выборки.
- Для выполнения команды необходимо обладать соответствующими правами. Команда может быть выдана либо пользователем, создавшим индекс, либо пользователем SYSDBA или имеющим аналогичные SYSDBA права.
- Команда ALTER INDEX неприменима к индексам, используемым в качестве ограничений логической целостности, определенным, как UNIQUE, PRIMARY KEY или FOREIGN KEY. Для модификации таких индексов следует использовать команду ALTER TABLE.
- ALTER INDEX неприменима также для изменения состава столбцов в индексе. Для выполнения подобной операции необходимо удалить индекс командой DROP INDEX и создать его снова командой CREATE INDEX

КомандаSETSTATISTICS

Синтаксис:

```
SET STATISTICS INDEX name;
```

В таблицах, где число повторений значений ключа индекса значительно меняется (возрастает или уменьшается), периодическая перекомпиляция индекса может значительно ускорить время обработки. SET STATISTICS повторно вычисляет селективность индекса. Индексная селективность рассчитывается исходя из количества различных значений ключа. Результаты размещаются в памяти и используются оптимизатором InterBase для построения плана обработки запроса. Сам индекс при этом не перестраивается. Для перестройки индекса следует пользоваться командой ALTER INDEX (точнее, парой ALTER INDEX name INACTIVE; ALTER INDEX name ACTIVE;).

Для выполнения команды необходимо обладать соответствующими правами. Команда может быть выдана либо пользователем, создавшим индекс, либо пользователем SYSDBA или имеющим аналогичные SYSDBA права.

Пример 4.23

```
SET STATISTICS INDEX TREADER_RDNUMB;
```

Команда DROP INDEX

Синтаксис:

```
DROP INDEX name;
```

Команда DROP INDEX удаляет существующий индекс. Для выполнения команды необходимо обладать соответствующими правами. Команда может быть выдана либо пользователем, создавшим индекс, либо пользователем SYSDBA или имеющим аналогичные SYSDBA права.

Индекс не может быть удален, пока он используется. Для "ждущих" транзакций (использующих опцию WAIT), выполнение команды будет отложено до завершения транзакций, в которых используется индекс. Для "нежданных" транзакций (использующих опцию NOWAIT), выполнение команды будет завершено немедленно с выдачей сообщения об ошибке.

Если индекс был создан автоматически как ограничение UNIQUE, PRIMARY KEY или FOREIGN KEY, команда DROP INDEX неприменима. Для удаления такого индекса можно воспользоваться командой ALTER TABLE с указанием соответствующей конструкции.

Пример 4.24

```
DROP INDEX TREADER_RDNUMB;
```

4.5. Исключения

Исключение - это поименованное сообщение об ошибке. Исключение может быть инициировано хранимой процедурой или триггером. Инициировано может быть только предварительно объявленное исключение. Раз объявленное исключение может быть использовано в любых хранимых процедурах и триггерах. Исключения хранятся в базе данных.

Исключения создаются командой CREATE EXCEPTION, изменяются командой ALTER EXCEPT и удаляются командой DROP EXCEPT. Инициируется исключение командой EXCEPTION exc_name. Инициированное исключение прерывает обработку и выдает сообщение об ошибке, если не была предусмотрена (заданием конструкции WHEN) ее специальная обработка. Текст выданного сообщения может быть получен приложением и обработан в нем.

Команда CREATE EXCEPTION

Синтаксис:

```
CREATE EXCEPTION name 'message';
```

name - имя исключения, по которому оно может быть инициировано.

'message' - текст сообщения об ошибке, связанный с исключением и выдаваемый при инициировании исключения.

Команда CREATE EXCEPTION создает новое исключение.

Пример 4.25

```
CREATE EXCEPTION NO_BOOKNM
  'Не указано наименование книги';
```

Команда ALTER EXCEPTION

Синтаксис:

```
ALTER EXCEPTION name 'message';
```

name - имя изменяемого исключения.

'message' - новый текст сообщения об ошибке, связанный с исключением и выдаваемый при инициировании исключения.

Команда ALTER EXCEPTION изменяет текст, связанный с существующим исключением. Исключение может быть изменено только его создателем.

Пример 4.26

```
ALTER EXCEPTION NO_BOOKNM
  'Не указано наименование книги!!';
```

Исключение может быть изменено даже в том случае, если оно используется в хранимых процедурах и триггерах (они ссылаются на его имя, а команда ALTER EXCEPTION меняет не имя, а только текст, так что изменение исключения не оказывает на них прямого влияния).

Команда DROP EXCEPTION

Синтаксис:

```
DROP EXCEPTION name;
```

name - имя удаляемого исключения.

Команда DROP EXCEPTION удаляет существующее исключение. Исключение может быть удалено только его создателем.

Пример 4.27

```
DROP EXCEPTION NO_BOOKNM;
```

В отличие от команды ALTER EXCEPTION, исключение не может быть удалено, если оно используется в хранимых процедурах и триггерах (удаление исключения делает ссылку на его имя в процедурах и триггерах неразрешенной). Нельзя также удалить исключение, если оно используется активной транзакцией.

Глава 5

Триггеры и хранимые процедуры

В базе данных могут храниться не только данные и их описания, но и программы, обеспечивающие работу с ними. К таким программам относятся триггеры и хранимые процедуры.

5.1. Триггеры и их назначение

Триггер - это отдельная хранимая в базе подпрограмма, связанная с таблицей или обзором, которая автоматически включается, когда в таблицу или обзор вставляется (триггер добавления), модифицируется (триггер модификации) или удаляется (триггер удаления) строка.

Триггер никогда не вызывается непосредственно. Он выполнятся всякий раз, когда приложение или пользователь пытаются вставлять, модифицировать или удалять строку в таблице. Другими словами триггер жестко связан с данными и выполняется тогда и только тогда, когда делается попытка изменить данные.

Триггеры могут использовать исключения (генерируемые сообщения об ошибках). Когда в триггере создается исключение, его работа завершается, отменяются все сделанные в триггере изменения и генерируется сообщение об ошибке, если не предусмотрена специальная обработка возникших ошибок (см. конструкцию WHEN).

Триггеры позволяют:

- Контролировать входные данные, обеспечивая повышение достоверности информации и ее логическую непротиворечивость.
- Повысить независимость прикладного программного обеспечения. Изменение схемы контроля в триггере автоматически отражается

во всех приложениях, не требуя ни внесения в них каких-либо изменений, ни их перетрансляции.

- Обеспечить автоматическую регистрацию изменений в таблицах. Приложение может хранить полный протокол изменений, используя триггеры, которые включаются при каждом изменении таблицы.
- Выполнять синхронные изменения в нескольких таблицах, обеспечивая как логическую целостность данных, так и автоматическое поддержание соответствия первичных и агрегированных данных.
- Автоматически уведомлять об изменениях в базе данных, используя события, создаваемые триггерами.

Триггеры создаются командой CREATE TRIGGER, модифицируются командой ALTER TRIGGER и удаляются командой DROP TRIGGER.

Триггер состоит из заголовка и тела. Заголовок содержит:

- Имя триггера, уникальное в пределах базы данных.
- Имя таблицы, для которой создается триггер.
- Действия с таблицей, при наступлении которых триггер включается.

Тело триггера содержит:

- Необязательный список локальных переменных с указанием их типов.
- Программный блок на языке процедур и триггеров INTERBASE (набор инструкций в операторных скобках BEGIN и END). Программный блок выполняется при включении триггера. Блок может включать в себя другие блоки, так что программа, реализующая триггер может быть сколь угодно сложной. Описание SQL для хранимых процедур и триггеров приведено ниже.

5.2. Хранимые процедуры и их назначение

Хранимая процедура - отдельная программа, написанная на SQL для процедур и триггеров InterBase. Сами процедуры хранятся в базе данных. Хранимые процедуры позволяют вести поиск и обработку данных непосредственно на сервере, обеспечивая максимальную независимость клиентской части приложений. В них могут использоваться любые конструкции SQL для процедур и триггеров (см. следующий раздел), кроме контекстных переменных NEW.*column*, OLD.*column*, применимых только в триггерах. Они, как обычные программы, могут получать входные параметры и возвращать значения вызвавшим их

приложениям. Кроме того, могут возвращать не только отдельный набор значений - строку, но и множество строк, которое можно рассматривать как виртуальную таблицу.

Хранимая процедура может также вызываться непосредственно из приложения или других хранимых процедур или триггеров. Хранимые процедуры, возвращающие множество строк, можно использовать в команде SELECT на месте таблиц или обзоров.

Использование хранимых процедур дает ряд преимуществ:

- *Модульность проектирования.* Приложения, которые обращаются к одной базе данных, могут совместно использовать хранимые процедуры, устраняя двойной код, уменьшая размер приложений и устраняя потенциальные ошибки.
- *Локализация изменений.* Если процедура модифицируется, то все внесенные изменения автоматически отражаются во всех приложениях, которые используют процедуру, обеспечивая их согласованность. При этом нет необходимости в перетрансляции и перекомпоновке приложений.
- *Ускорение обработки.* Хранимые процедуры выполняются сервером, а не клиентом, что позволяет ускорить обработку запросов и сократить сетевой трафик. Последнее особенно важно для удаленного клиентского доступа.

Процедуры по своему назначению разделяются на два вида: *выполнимые процедуры* и *процедуры выбора*.

Выполнимая процедура - это обычная программа, которая получает несколько (возможно, и ноль) параметров, выполняет какие-либо действия в базе данных и возвращает несколько (возможно, и ноль) значений.

Процедура выбора - это программа, которая получает несколько (возможно, и ноль) параметров, выполняет какие-либо действия в базе данных и возвращает множество (возможно, пустое) наборов значений. Другими словами, процедура выбора создает вычисляемую таблицу, хотя такая таблица никуда и не записывается. Это позволяет обращаться к процедуре, как к таблице, используя команду SELECT.

Синтаксически можно обращаться к выполнимой процедуре, как к процедуре выбора, получая при этом в ответ в точности одну строку, а к процедуре выбора, как к выполнимой, получая при этом в ответ первую строку формируемой таблицы. Ценность такой инверсии обращений в лучшем случае нулевая, поэтому, несмотря на отсутствие прямого запрета ее использования, нет оснований для практического применения инверсных вызовов хранимых процедур.

5.3. SQL для триггеров и хранимых процедур в InterBase

SQL для триггеров и хранимых процедур в InterBase представляет собой законченный язык программирования для манипулирования данными.

Язык включает:

- инструкции манипуляции данных SQL: добавление, модификация, удаление из базы, выборка данных из базы в список переменных.
- операторы SQL и выражения, включая функции пользователя (UDF - user defined functions).

расширение SQL, включающее оператор присвоения, операторы управления последовательностью вычислений, возможность использования собственных и контекстных переменных, операторы генерации событий и исключений (ошибок), а также команды обработки ошибок.

При работе с данными используются следующие операции (перечень дан в порядке убывания их приоритета):

- операция конкатенации (объединения) для строковых данных
  - "||"
- арифметические операции
  - "* /"
  - "+ _"
- операции сравнения
  - = ==
  - o, !=, ~=, ^= (не равно)
  - >
  - <
  - >=
  - <=
  - !>, ~>, ^> (не больше)
  - !<, ~<, ^< (не меньше)
- логические операции
  - NOT
  - AND
  - OR

Хотя хранимые процедуры и триггеры используются различным образом и в разных целях, они базируются на одном и том же языке. И хранимые процедуры, и триггеры могут использовать любые конструкции языка, за исключением следующих:

- Контекстные переменные допустимы только в триггерах.

Входные и выходные параметры, а также инструкции SUSPEND и EXIT, которые возвращают значения, применимы только в хранимых процедурах.

Прежде чем продолжить, уточним терминологию.

Оператор DECLARE - это оператор объявления переменных.

Блок (< block>) - это один или несколько операторов (<compound_statement>), заключенных в операторные скобки BEGIN-END.

Оператор (<compound_statement>) - это простой оператор (statement) или блок.

Формализованная запись:

```
< block> .• :=
BEGIN
< compound_statement>
[<compound_statement>...]
END
< compound_statement> ::=
{< Ыоск> | statement;}
```

Простые операторы: оператор присвоения, оператор генерации исключения, оператор вызова процедуры, оператор ветвления IF, оператор цикла FOR, оператор цикла WHILE, оператор генерации события POST_EVENT, операторы SQL INSERT, UPDATE, SELECT, оператор возврата значений выходных параметров SUSPEND, оператор прерывания процедуры EXIT, оператор обработки ошибок WHEN.

Кроме того, для удобства сопровождения программ в их текст могут быть внесены комментарии.

Рассмотрим подробнее перечисленные операторы.

ОПЕРАТОР ПРИСВОЕНИЯ

Синтаксис:

```
variable = < expression>;
```

variable - локальная переменная, входной или выходной параметр, контекстная переменная.

<expression> - любое допустимое в SQL выражение, включающее переменные, операторы SQL, пользовательские (UDF) функции и генераторы, выражения в скобках.

ОПЕРАТОР ОБЪЯВЛЕНИЯ ПЕРЕМЕННЫХ DECLARE

Оператор DECLARE применяется в хранимых процедурах и триггерах и размещается в начале их тела непосредственно перед скобкой BEGIN, за которой размещаются все исполнимые операторы. Все используемые переменные должны быть объявлены. Оператор объявления завершается ";". Одним оператором может быть объявлена только одна переменная, и объявления имеют смысл только внутри хранимой процедуры или триггера.

Синтаксис:

```
DECLARE VARIABLE var datatype;
```

ОПЕРАТОР ГЕНЕРАЦИИ ИСКЛЮЧЕНИЯ

Генерирует сообщение об ошибке **(исключение). Применяется в** хранимых процедурах и триггерах.

Синтаксис:

```
EXCEPTION name;
```

name - имя генерируемого исключения. Исключение с данным именем должно быть предварительно создано в базе командой CREATE EXCEPTION.

ОПЕРАТОР ВЫЗОВА ПРОЦЕДУРЫ

Оператор осуществляет вызов **хранимой процедуры. Применяется в** хранимых процедурах и триггерах.

Синтаксис:

```
EXECUTE PROCEDURE name [LIST_: param./
[RETURNING_VALUES LIST_param];
```

name - имя вызываемой процедуры. Сама процедура должна быть предварительно создана в базе командой CREATE PROCEDURE.

LIST_: param ::= :param [, *LIST_:* param/, заданный после имени процедуры *name* - это список входных параметров процедуры (если процедура не требует параметров, то может отсутствовать). Отдельные параметры могут быть константами или переменными. Перед именем переменной ставится ":", кроме случаев использования контекстных переменных NEW и OLD. *LIST_*:param, заданный после RETURNING_VALUES– список возвращаемых процедурой значений (если процедура возвращает значения). Перед именем возвращаемых значений ставится ":". В триггерах и процедурах использование вызовов других

процедур аналогично использованию подпрограмм в традиционных алгоритмических языках.

Замечание. При вызове из ISQL или прикладных программ на базовых языках синтаксис вызова отличается от приведенного.

ОПЕРАТОР ВЕТВЛЕНИЯ

Оператор ветвления IF ... THEN ... ELSE обеспечивает выполнение того или иного действия в зависимости от истинности проверяемого условия.

Синтаксис:

```
IF ( <condition>) THEN
<compound_statement>
[ELSE
<compound_statement>]
```

<compound_statement> ::= {< block> | statement;} (см. выше)

<condition> - выражение, которое может принимать значение истина или ложь.

Если условие выполнено (значение TRUE), то выполняется оператор, следующий за конструкцией THEN, иначе выполняется оператор, следующий за конструкцией ELSE, если она присутствует.

Например,

```
. . .
IF (a<0) THEN
 b = -a;
ELSE
 b = a;
. . .
```

Оператор цикла FOR

Цикл FOR обеспечивает выполнение заданного оператора для каждой строки команды SELECT. Цикл FOR может использоваться в хранимых процедурах и триггерах.

Синтаксис:

```
FOR
< select_expr>
DO
< compound_statement>
```

FOR SELECT - инструкция цикла, которая **отыскивает** строку, указанную в <select_expr> и для каждой строки исполняет оператор или блок, указанный после ключевого слова DO.

Конструкция <select_expr> представляет собой обычную команду SELECT, к которой добавлена обязательная здесь опция INTO и которая должна быть последней опцией команды SELECT.

В опции INTO перечисляются имена локальных переменных или параметров процедуры, которым присваиваются соответствующие значения выбранных командой SELECT столбцов. Имена переменных должны предваряться символом ":".

В качестве примера возьмем фрагмент процедуры PAUTHOR.

Пример 5.1

```
FOR SELECT AUTHOR, AUNAME FROM TAUTHOR
  INTO :AUTHOR, :AUNAME
 DO
  IF(AUNAME>P1) THEN
  IF(AUNAME<P2) THEN SUSPEND;
```

Здесь оператор SUSPEND обеспечивает передачу вызывающему приложению данных, удовлетворяющих дополнительному условию.

ОПЕРАТОР ЦИКЛА WHILE

Оператор цикла WHILE обеспечивает выполнение оператора, указанного после ключевого слова DO пока указанное после WHILE условие истинно. Цикл WHILE может использоваться в хранимых процедурах и триггерах.

Синтаксис:

```
WHILE ( <condition>) DO
< compound_statement>
```

WHILE ... DO - оператор выполнения цикла, который повторяет оператор или блок < compound_statement>, указанный после DO, пока условие <condition> истинно. Условие проверяется в начале каждого цикла.

В качестве примера рассмотрим вычисление факториала.

Пример 5.2

```
S=1;
WHILE (I > 0) DO
BEGIN
S = S * I;
I = I - 1;
END
```

ОПЕРАТОР ГЕНЕРАЦИИ СОБЫТИЯ POST_EVENT

Оператор POST_EVENT используется для генерации события, которое может быть в дальнейшем обработано в приложениях.

Сама обработка событий в InterBase строится по следующей схеме.

1. Приложение выдает команду

```
EVENT INIT request_name (event_name1 [, event_name2 ...]);
```

По этой команде создается список событий request_name, содержащий имена событий event_name1 ,event_name2 ...

2. То же или другое приложение выдает команду

```
EVENT WAIT request_name;
```

По этой команде приложение приостанавливается и ожидает наступления одного из событий в списке request_name.

3. Само событие генерируется внутри триггера или хранимой процедуры командой POST_EVENT. После того как такое событие произошло, приложение, ожидающее событие, получает соответствующее сообщение и продолжает свою работу. Такой механизм позволяет приложениям обрабатывать различные специфические ситуации при работе с базой данных.

Внутри триггеров и хранимых процедур реализуется только команда POST_EVENT.

Синтаксис:

```
POST_EVENT <event_name>;
```

Параметр <event_name> может быть либо символьным литералом в кавычках, либо строковой переменной.

Замечание. Имена переменных в хранимых процедурах не должны предваряться символом ":" нигде, кроме как в командах SELECT, INSERT, UPDATE, DELETE, что позволяет отличать их от имен столбцов.

При выполнении процедуры команда POST_EVENT сообщает диспетчеру событий о наступлении события. Диспетчер событий следит за приложениями, ждущими событий, и извещает об их наступлении.

Пример 5.3

```
POST_EVENT "Oh_oh_oh";
```

или

```
ABC= "Oh_oh_oh";
POST_EVENT ABC;
```

ОПЕРАТОРЫ SQL INSERT, UPDATE, SELECT, DELETE

Внутри хранимых процедур и триггеров могут использоваться стандартные команды SQL: INSERT, UPDATE, SELECT, DELETE. Единственной особенностью этих команд внутри процедур и триггеров является то, что в них могут использоваться в качестве параметров локальные переменные процедур. Для того чтобы отличать локальные переменные от столбцов таблиц в командах INSERT, UPDATE, SELECT, DELETE, имена локальных переменных предваряются символом ":".

Кроме того, для помещения результатов выборки в локальные переменные к команде SELECT добавляется опция INTO : var [,: var ...].

ОПЕРАТОР ВОЗВРАТА ЗНАЧЕНИЙ ВЫХОДНЫХ ПАРАМЕТРОВ SUSPEND

Оператор SUSPEND предназначен для использования в хранимых процедурах (в триггерах SUSPEND неприменим), причем только в процедурах выбора, хотя синтаксически допустим и в выполнимых процедурах.

Оператор SUSPEND приостанавливает выполнение процедуры выбора, возвращает управление вызвавшей программе и возобновляет работу со следующей команды, когда выполнена очередная команда FETCH. SUSPEND возвращает вызвавшей программе результаты работы процедуры в выходных параметрах.

SUSPEND не должен использоваться в выполнимых процедурах, так как команды, следующие за ним, никогда не будут выполнены. В выполнимых процедурах следует использовать оператор явного выхода EXIT.

В процедуре выбора команда SUSPEND возвращает текущие значения выходных параметров вызвавшей программе и продолжает выполнение. Если какой-либо выходной параметр не получил значение явно, то его содержимое непредсказуемо, что может привести к ошибкам, поэтому процедура должна обязательно присваивать значения всем выходным параметрам перед выполнением SUSPEND.

ОПЕРАТОР ПРЕРЫВАНИЯ ПРОЦЕДУРЫ EXIT

И в процедурах выбора и в выполнимых процедурах оператор EXIT передает управление на конец процедуры (завершающий END).

Действие, выполняемое по достижении конца процедуры, зависит от ее типа:

- В процедуре выбора конечная команда END возвращает управление вызвавшему ее приложению и устанавливает SQLCODE в 100; это указывает, что список найденных ею строк закончен.

В выполнимой процедуре конечная команда END возвращает управление вызвавшему ее приложению с установкой значений выходных параметров, если они есть.

Сводка результатов выполнения операторов SUSPEND, EXIT и END приведена в таблице.

Таблица 5.1. *Операторы приостановки - завершения процедуры*

Тип процедуры	*SUSPEND*	*EXIT*	*END*
Процедура выбора	Приостанавливает работу до выполнения очередной команды FETCH и возвращает значения выходных параметров	Переходит к завершающему процедуру оператору END	Возвращает управление вызвавшему ее приложению и устанавливает SQLCODE в 100 (конец потока)
Выполнимая процедура	Переходит к завершающему процедуру оператору END. Не рекомендуется	Переходит к завершающему процедуру оператору END	Возвращает значения и передает управление вызвавшему ее приложению

ОПЕРАТОР ОБРАБОТКИ ОШИБОК WHEN

Оператор WHEN ... DO обеспечивает обработку возникших ошибок. Оператор применяется в хранимых процедурах и триггерах.

Синтаксис:

```
WHEN {LIST_< error> / ANY}
DO <compound_statement>

< error> ::=
{EXCEPTION exception_name / SQLCODE number / GDSCODE errcode}
```

Оператор WHEN должен быть последним в блоке BEGIN...END. С его помощью процедуры и триггеры могут обрабатывать ошибки трех типов:

- Исключения, инициированные оператором EXCEPTION в данной процедуре или процедурах, прямо или косвенно вызванных данной, а также исключения, инициированные в триггерах, вызванных в результате действий этих процедур.
- SQL-ошибки, идентифицирующиеся SQLCODE.

Ошибки, идентифицирующиеся кодами ошибок InterBase.

Конструкция ANY позволяет выполнять операторы обработки при возникновении любых ошибок перечисленных типов.

Сводка синтаксиса оператора WHEN приведена в таблице.

Таблица 5.2. *Синтаксические конструкции оператора WHEN*

Параметр	***Описание***
EXCEPTION exception_name	Имя исключения (описанного в базе)
SQLCODE number	Код ошибки - SQLCODE
GDSCODE errcode	Код ошибки InterBase
ANY	Обеспечивает вызов обработчика для любых перехватываемых ошибок
<compound_statement>	Простой оператор или блок, осуществляющий обработку ошибок

Обработка исключений

Вместо завершения работы при возникновении исключения процедура может обработать и возможно исправить ситуацию, приведшую к исключению. При возникновении исключения выполняются следующие действия:

- Прекращается выполнение блока BEGIN ... END, содержащего исключение, и отменяются действия, выполненные в блоке.
- Если блок содержит оператор WHEN, управление передается в WHEN, в противном случае аналогичные действия производятся в блоке, содержащем данный. Подобные действия производятся до тех пор, пока либо не будет найден соответствующий WHEN, либо не будет достигнут уровень процедуры. В последнем случае процедура будет завершена, а исключение не будет обработано.
- Выполняются действия, определенные оператором (блоком), заданным в конструкции WHEN (если он обнаружен).

Управление возвращается оператору (блоку) программы, следующему за оператором WHEN.

Если исключение обработано с помощью WHEN, то соответствующее сообщение об ошибке не выдается.

Обработка ошибок SQL

Процедуры могут также обрабатывать ошибки SQL по кодам, возвращенным в SQLCODE. После выполнения каждой команды SQL формируется код ее завершения - SQLCODE, отражающий успешность выполнения, или код ошибки. SQLCODE может также содержать код предупреждения, типа того, что перечень строк в выборке по циклу FOR SELECT исчерпан.

Таблица 5.3. *Коды завершения команд SQL*

SQLCODE	*Описание*
0	Успешное завершение
1-99	Предупреждение или информационное сообщение
100	Конец файла (списка)
<0	Ошибка. Команда не выполнена

Подробный перечень значений SQLCODE приведен в Приложении Б.

Пример 5.4

```
WHEN SQLCODE -803
/*Попытка добавить строку со значением первичного ключа,
которое уже есть в таблице */
DO
BEGIN
. . .
```

Обработка ошибок InterBase

Процедуры могут обрабатывать ошибки InterBase по кодам, возвращенным в GDSCODE. Например, если команда в процедуре пытается модифицировать строку, уже модифицированную другой, еще не завершенной, транзакцией, то в этом случае процедура могла бы получать код ошибки InterBase, isc_lock_conflict. При повторении попытки ее модификации, другая транзакция может выполнить откат, сняв, таким образом, блокировку, что позволит успешно завершить команду. Используя инструкцию WHEN GDSCODE, процедура может обрабатывать ошибки конфликта блокировки и повторять его операцию.

В качестве примера рассмотрим фрагмент хранимой процедуры с возвращаемым параметром RETCODE, который устанавливается в 0 при нормальном выполнении процедуры и в 1 при возникновении каких либо ошибок.

Пример 5.5

```
. . .
BEGIN
 RETCODE=0;
 . . .
 WHEN ANY DO
 BEGIN
  RETCODE=1;
  . . .
  EXIT;
 END
END
```

Комментарий

Комментарий - произвольный текст, который может быть размещен в любом месте процедуры или триггера.

Синтаксис:

```
/* текст ...

*/
```

Текст комментария может занимать несколько строк.

В следующей таблице дана краткая сводка операторов SQL для процедур и триггеров.

Таблица 5.4. *Сводка операторов SQL для процедур и триггеров*

Команда	*Описание*
BEGIN... END	Операторные скобки, задающие составной оператор (блок). BEGIN - ключевое слово, идентифицирующее начало блока; END - ключевое слово, идентифицирующее конец блока. Внутри блока может быть любое количество операторов. Ни после BEGIN, ни после END точка с запятой не ставится
variable = expression	Оператор присваивания. Задает новое значение (выражение expression) локальной, контекстной переменной или полю базы данных
/* comment_text */	Комментарий программиста. Комментарий может занимать несколько строк. На выполнение программы не влияет

Команда	*Описание*
EXCEPTION exception_name	Инициирует исключение. Исключение - определяемая пользователем ошибка. Исключение возвращает вызвавшему приложению сообщение об ошибке, если оно не обработано инструкцией WHEN
EXECUTE PROCEDURE proc_name *[LIST_var]* *[*RETURNING_VALUES; *LIST_*var*]*	Выполняет хранимую процедуру. proc_name - имя процедуры, за которым указывается список параметров процедуры и список возвращаемых значений (после ключевого слова RETURNINGJVALUES). Входные и выходные параметры представляют собой локальные переменные
FOR <select_statement>;DO <compound_statement>	Задает цикл выполнения оператора или блока (группы операторов в операторных скобках BEGIN-END) для всех строк получаемых по запросу, определенному конструкцией <select_statement>. <select_statement> - обычная команда SELECT, за исключением обязательной здесь опции INTO. Поле INTO указывает список локальных переменных, в которые помещаются данные из очередной выбранной по запросу строки
<compound_statement>	Простой оператор или блок (группа операторов в операторных скобках BEGIN-END)
IF (<condition>);THEN <compound_statement>; /ELSE <compound_statement>*]*	Проверяет условие <condition> и в зависимости от его истинности выполняет либо оператор (блок), следующий за ключевым словом THEN в случае истинности условия, либо оператор (блок), следующий за ключевым словом ELSE в случае ложности условия. Конструкция ELSE является необязательной
NEW.column	Контекстная переменная. Указывает на значение, которое предполагается присвоить полю базы данных (используется в триггерах операций вставки и обновления)
OLD.column	Контекстная переменная. Указывает на значение, которое имело поле базы данных до выполнения над ним операций удаления или обновления (используется в триггерах операций вставки и обновления)

Команда	*Описание*
POST_EVENT event_name	Инициирует событие event_name
WHILE (<condition>) DO <compound_statement>	Задает цикл выполнения оператора или блока (группы операторов в операторных скобках **BEGIN-END**) <compound_statement> до тех пор, пока истинно условие <condition>. Если условие изначально ложно, то цикл не вьшолняется ни разу
WHEN *{LIST_*<error>*}*|ANY*}* DO <compound_statement>	Оператор обработки сообщений об ошибках. Если происходит одно из списка событий <error>, то выполняется оператор (блок) <compound_statement>. Если необходимо использовать оператор WHEN, то он должен быть помещен в конце блока, непосредственно перед операторной скобкой END. Конструкция <error> имеет следующий вид: EXCEPTION exception_name, где exception_name - имя исключения; SQLCODE errcode, где errcode код ошибки SQL; GDSCODE number, где number номер ошибки InterBase; ANY - любые исключения или ошибки

5.4. Команды создания, удаления, модификации триггеров; работа с ними

Команда CREATE TRIGGER

Синтаксис:

```
CREATE TRIGGER name FOR { table / view}
 [ACTIVE / INACTIVE;
 {BEFORE / AFTER} {DELETE / INSERT / UPDATE;
 [POSITION number;
AS < trigger_body>

< trigger_body> ::=
 [< variable_declaration_list>]
< block>

< variable_declaration_list> ::=
```

```
DECLARE VARIABLE variable datatype;
[< variable_declaration_list>]

< block> ::=
BEGIN
<L_statement>
END

<L_statement> ;;=
< compound_statement>
[<L_statement>]

< compound_statement> = {< block> / statement;}
```

Таблица 5.5. *Синтаксические конструкции команды CREATE TRIGGER.*

Описание	***Параметр***
Name	Имя триггера. Имя должно быть уникальным в базе данных
table | view	Имя таблицы, для которой создается триггер
ACTIVE | INACTIVE	Необязательная конструкция. Определяет активность триггера. ACTIVE - триггер включен, INACTIVE - триггер отключен
BEFORE | AFTER	Обязательный. Определяет, когда включается триггер: BEFORE - перед операцией, выполняемой над таблицей. AFTER - после операции, выполняемой над таблицей
DELETE | INSERT | UPDATE	Указывает, при выполнении какой именно операции с таблицей будет включаться триггер: DELETE - удаление, INSERT -вставка, UPDATE - модификация
POSITION number	POSITION number задает порядок, в котором будут выполняться триггеры (с одной таблицей и одними и теми же условиями включения может быть связано несколько триггеров). number должен быть целым числом между 0 (по умолчанию) и 32767 включительно. Включение триггеров происходит в порядке возрастания номеров (number). Нумерация может и не быть последовательной. Если триггеры имеют один и тот же номер, то они будут включаться в алфавитном порядке их имен

Описание	*Параметр*
DECLARE VARIABLE variable datatype;	Объявляет локальные переменные, используемые только в триггере. Объявление каждой переменной начинается с ключевых слов DECLARE VARIABLE и заканчивается ";". Имена объявляемых переменных должны быть уникальными в триггере. datatype задает тип локальной переменной
<compound_statement>	Оператор SQL для хранимых процедур и триггеров. Оператор завершается ";", если это не блок BEGIN-END. После END ";" не ставится

Рассмотрим триггер для контроля добавления данных в нашу тестовую таблицу авторов. Наличие подобных триггеров обеспечивает полноту и корректность строк таблиц.

Пример 5.6

```
CREATE TRIGGER I_TAUTHOR_1 FOR TAUTHOR
ACTIVE BEFORE INSERT POSITION 0
as
begin
  if (new.AUTHOR is NULL) then
    new.AUTHOR=GEN_ID(sysnumber,1);
  if (new.AUNAME is NULL) then exception NO_AUTHOR;
  if (new.COMMENT is NULL) then new.COMMENT=" ";
end
```

Здесь проверяется заполнение ключа таблицы AUTHOR, **имени** автора AUNAME и комментария COMMENT. При отсутствии имени автора генерируется исключение (сообщение об ошибке). При отсутствии ключа таблицы AUTHOR генерируется новое уникальное значение и присваивается ключу. При отсутствии комментария COMMENT значение комментария устанавливается в пробел.

В триггере обновления для таблицы книг осуществляется проверка на корректность связей (в данном примере - ссылки на рубрику) между строками таблиц.

Пример 5.7

```
CREATE TRIGGER I_TBOOK_1 FOR TBOOK
ACTIVE BEFORE INSERT POSITION 0
as
begin
  if (new.UNIKEY is NULL) then
   new.UNIKEY=GEN_ID(sysnumber,1);
  if (new.MATHERKEY is NULL or new.MATHERKEY<0) then
```

Триггеры и хранимые процедуры 129

```
    exception NO_RUBRIC;
  if (new.MATHERKEY>0) then
   if (NOT EXISTS (select * from TBOOK where
       (unikey=new.MATHERKEY)))
    then exception ERR_RUBRIC;
  if (new.BOOKNM is NULL) then exception NO_BOOKNM;
end
```

Команда ALTER TRIGGER

Команда ALTER TRIGGER предназначена для изменения ранее созданных триггеров. Триггер может быть изменен только его создателем, SYSDBA или другим пользователем с аналогичными правами.

Используя команду ALTER TRIGGER, можно изменить заголовок триггера, тело триггера и то и другое одновременно. В заголовке триггера можно изменить: признак активности триггера, условие включения триггера (AFTER-BEFORE, INSERT-UPDATE-DELETE), порядок включения триггеров. Изменение тела триггера состоит в его замене на вновь указанное.

Кроме того, триггеры могут создаваться системой автоматически по команде CHEK. Для изменения таких триггеров следует использовать команду ALTER TABLE.

Команда ALTER TRIGGER может быть также использована для отключения триггера без его удаления. Если установить опцию INACTIVE, то триггер не будет включаться. В то же время он сохраняется в базе и для его повторного включения достаточно выдать ALTER TRIGGER с опцией ACTIVE.

Синтаксис:

```
ALTER  TRIGGER name
[ACTIVE / INACTIVE;
[{BEFORE / AFTER; {DELETE / INSERT / UPDATE}]
/•POSITION number;
AS < trigger_body>;
```

Синтаксис ALTER TRIGGER аналогичен синтаксису CREATE TRIGGER, за исключением:

- В команде отсутствует опция указания таблицы, к которой относится триггер. При создании триггера он связывается с определенной таблицей и эта связь не может быть изменена, иначе это будет уже другой триггер, для создания которого нужна другая команда CREATETRIGGER.
- Если какие-либо опции в ALTER TRIGGER отсутствуют, то они копируются из существующего описания триггера.

Изменение заголовка триггера

При необходимости изменить заголовок триггера надо задать список изменяемых опций триггера (хотя бы одной опции). При этом остальные опции триггера не будут меняться. Если меняются опции AFTER или BEFORE, то должна быть указана и соответствующая им опция из перечня INSERT, UPDATE, DELETE.

Пример 5.8

```
AFTER TRIGGER I_TBOOK_1 INACTIVE POSITION 12;
```

Изменение тела триггера

При необходимости изменить только тело триггера в заголовочной части следует задавать только имя триггера.

Тело триггера, заданное в команде AFTER TRIGGER, заменяет тело существующего триггера. На практике можно рекомендовать предварительно выбрать тело существующего триггера, затем внести в него изменения и загрузить командой AFTER TRIGGER. Конкретная последовательность действий зависит от того, какие средства используются для работы с базой. В состав стандартной поставки для работы в среде Windows входит утилита WISQL. В то же время существует большое количество утилит, значительно более удобных в работе.

В качестве примера модифицируем тело триггера I_TAUTHOR_1, задав в нем новое значение для столбца COMMENT _"***".

Пример 5.9

```
AFTER TRIGGER I_TAUTHOR_1
as
begin
  if (new.AUTHOR is NULL) then
new.AUTHOR=GEN_ID(sysnumber,1);
  if (new.AUNAME is NULL) then exception NO_AUTHOR;
  if (new.COMMENT is NULL) then new.COMMENT="***";
end
```

Команда DROP TRIGGER

Команда DROP TRIGGER используется для физического удаления триггера из базы данных. Для временного отключения триггера следует использовать команду AFTER TRIGGER. Автоматически созданные триггеры (конструкцией CHECK) не могут удаляться. Для удаления таких триггеров следует использовать команду ALTER TABLE. Триггер может быть удален только его создателем, SYSDBA или другим пользователем с аналогичными правами и только, если он в данное время никем не используется.

Синтаксис:

```
CROP TRIGGER name;
```

Пример 5.10

(Лучше этот пример не исполнять: триггера жалко - он хороший.)

```
DROP TRIGGER I_TAUTHOR_1;
```

5.5. Команды создания, удаления, модификации хранимых процедур; работа с ними

СОЗДАНИЕ ХРАНИМЫХ ПРОЦЕДУР

Хранимые процедуры создаются командой CREATE PROCEDURE в ISQL. Во внедренном SQL команда недоступна.

Хранимая процедура состоит из заголовка и тела.

Заголовок хранимой процедуры содержит:

- Имя процедуры, которое должно быть уникальным среди имен процедур, таблиц и обзоров в базе данных.
- Необязательный список параметров процедуры, передаваемых процедуре вызывающими приложениями, с указанием их типов данных.

Для процедур, возвращающих значения вызывающим приложениям, необходимо указать список возвращаемых параметров с указанием их типов. Список указывается после ключевого слова RETURNS.

Тело процедуры содержит:

- Необязательный список локальных переменных с указанием их типов.

Блок операторов на языке процедур и триггеров, заключенный в операторные скобки BEGIN-END. Сам блок может содержать внутри себя Другие блоки. Количество вложений блоков практически не ограничено.

При использовании для ввода команды Script-файла необходимо помнить, что в соответствии с синтаксисом операторы внутри процедуры отделяются друг от друга символом ";". В то же время команды в Script-файле также отделяются ";". Чтобы устранить неоднозначность разделения команд, необходимо установить какой-либо иной ограничитель тек-

ста команд. Для этого следует непосредственно перед заданием команды CREATE TRIGGER выполнить команду SET TERM, устанавливающую такой ограничитель, а в конце работы той же командой SET TERM восстановить стандартный (";")•

Синтаксис команды CREATE PROCEDURE:

```
CREATE PROCEDURE name [(LIST_<params>)]
[RETURNS (LIST_<params>)]
AS
<procedure_body>;

LIST_<params> ::= param datatype

<procedure_body> ::=
[ <variable_declaration_list>]
<block>

<variable_declaration_list> ::=
DECLARE VARIABLE var datatype;
[<variable_declaration_list>]

<block> ::=
BEGIN
<L_statement>
END

<L_statement> ::=
< compound_statement>
[<L_statement>]

< compound_statement> = {< block> / statement;}
```

Таблица 5.5. *Синтаксические конструкции команды CREATE PROCEDURE*

Конструкция	*Описание*
name	Имя процедуры. Должно быть уникальным среди имен процедур, таблиц и обзоров в базе данных
param <datatype>	Параметр процедуры, передаваемый ей вызывающими приложениями. param - имя параметра, уникальное среди имен параметров и переменных процедуры. <datatype> - допустимый в InterBase тип данных

Конструкция	*Описание*
RETURNS param <datatype>	Выходные значения (параметры) процедуры, возвращаемые ею вызывающим приложениям. param - имя выходного параметра, уникальное среди имен параметров и переменных процедуры. <datatype> - допустимый в InterBase тип данных. Процедура возвращает значения выходных параметров вызывающим приложениям по достижении ее конца или по команде SUSPEND в теле процедуры
AS	Ключевое слово, отделяющее заголовок процедуры от ее тела
DECLARE VARIABLE var <datatype>	Объявляет локальные переменные, используемые в процедуре. Каждое объявление должно начинаться с ключевых слов DECLARE VARIABLE и заканчиваться ";". Один оператор объявляет одну переменную. var - имя переменной, уникальное среди имен параметров и переменных процедуры. <datatype> - допустимый в InterBase тип данных
statement	Любой допустимый на языке процедур и триггеров оператор. Оператор должен оканчиваться символом ";" (кроме операторных скобок BEGIN-END)

Перечень допустимых операторов в теле процедуры и их синтаксис описан в разд. 5.3.

Пример 5.11

Процедура, которая возвращает список книг, генерирует по таблице авторов для каждой книги полный список ее авторов через запятую.

```
CREATE PROCEDURE PBUTHOR(
    CODE INTEGER
)
RETURNS (
    AUTHORS VARCHAR(250)
)
AS
 declare variable auname varchar(60);
 declare variable UNIKEY integer;
 declare variable ws integer;
begin
```

```
 ws=-1;
 for select a.UNIKEY, b.auname
  from tbook a, tauthor b, tbook_author c
  where (a.unikey=:Code and a.unikey=c.bookkey and
c.author=b.author)
  into :UNIKEY, :auname
  do
   begin
    if(ws=-1) then authors=auname;
    else authors=authors||', '||auname;
    WS=UNIKEY;
   end
 if(ws!=-1) then suspend;
end
```

ИЗМЕНЕНИЕ ХРАНИМЫХ ПРОЦЕДУР

Изменение хранимых процедур осуществляется командой ALTER PROCEDURE. Команда ALTER PROCEDURE выполняется в ISQL. Во внедренном SQL команда недоступна.

ALTER PROCEDURE изменяет описание хранимой процедуры при условии сохранения корректности ссылок на нее со стороны объектов базы данных. Изменения, внесенные в хранимую процедуру, сразу становятся доступными для всех использующих ее приложений. Перекомпиляция соответствующих приложений не требуется.

Команда ALTER PROCEDURE может быть выполнена создателем процедуры, пользователем SYSDBA или другим пользователем с аналогичными правами.

При изменении хранимой процедуры новое описание замещает существующее. Необходимо помнить, что изменение состава, последовательности и типов входных и выходных параметров процедур, как и интерфейса прикладных программ, может повлечь необходимость внесения изменений в вызывающих программах.

На практике можно рекомендовать предварительно выбрать существующую хранимую процедуру, затем внести в нее изменения и загрузить командой AFTER PROCEDURE. Конкретная последовательность действий зависит от того, какие средства используются для работы с базой. В состав стандартной поставки для работы в среде Windows входит утилита WISQL. В то же время существует большое количество утилит, значительно более удобных в работе.

Синтаксис команды ALTER PROCEDURE аналогичен синтаксису команды CREATE PROCEDURE:

```
ALTER PROCEDURE name [(LIST_<params>)]
[RETURNS (LIST_<params>)]
```

```
AS
<procedure_body>;
```

name - имя существующей хранимой процедуры.

Состав параметров и опций команды полностью аналогичен параметрам и опциям команды CREATE PROCEDURE (см. разд. *Создание хранимых процедур)*.

Пример 5.12

```
ALTER PROCEDURE PBUTHOR(
    CODE INTEGER
)
RETURNS (
    AUTHORS VARCHAR(250)
)
AS
 declare variable auname varchar(60);
 declare variable UNIKEY integer;
 declare variable ws integer;
begin
 ws=-1;
 for select a.UNIKEY, b.auname
  from tbook a, tauthor b, tbook_author c
  where (a.unikey=:Code and a.unikey=c.bookkey and
c.author=b.author)
  into :UNIKEY, :auname
  do
   begin
    if(ws=-1) then authors=auname;
    else authors=authors||' - '||auname;
/* в исходном тексте разделителем в списке
   была запятая */
    ws=UNIKEY;
   end
 if(ws!=-1) then suspend;
end
```

УДАЛЕНИЕ ХРАНИМЫХ ПРОЦЕДУР

Удаление хранимых процедур осуществляется командой **DROP** PROCEDURE, которая выполняется в ISQL. Во внедренном SQL команда недоступна.

Команда DROP PROCEDURE удаляет существующую в базе данных процедуру, и может быть выполнена создателем процедуры, пользователем SYSDBA или другим пользователем с аналогичными правами.

Для успешного выполнения команды DROP PROCEDURE необходимо выполнение ряда условий. Нельзя удалить процедуру, если она ис-

пользуется другими процедурами, триггерами или обзорами базы данных. Перед удалением процедуры необходимо внести соответствующие изменения в объекты, использующие удаляемую процедуру. Нельзя удалить процедуру, если она используется незавершенной транзакцией. Необходимо завершить транзакцию и только после этого удалить процедуру.

Команда DROP PROCEDURE недоступна во внедренном SQL, необходимо использовать динамический SQL.

Синтаксис команды:

```
DROP PROCEDURE name;
```

name - имя существующей процедуры.

Пример 5.13

```
DROP PROCEDURE PBUTHOR;
```

ИСПОЛЬЗОВАНИЕ КОМАНД ALTER PROCEDURE И DROP PROCEDURE

При внесении изменений в процедуры или их удалении необходимо помнить, что в базе хранятся ссылки на них, если они используются другими объектами базы. В этом случае их изменение должно быть запрещено во избежание разрушения логической целостности базы.

Если процедура используется внешним приложением, то целостность приложений не может контролироваться средствами базы, поэтому, прежде чем вносить изменения, необходимо убедиться в отсутствии нежелательных последствий от таких изменений.

Изменения желательно проводить, когда с базой не работают приложения, использующие процедуры. В противном случае нельзя быть уверенным, с какой именно версией процедуры работают приложения.

При выполнении запросов на работу процедуры последняя помещается в кэш метаданных и остается там до завершения запроса. Изменения в процедуре становятся видимыми на клиенте после его отсоединения и повторного соединения с базой.

ОБРАЩЕНИЕ К ВЫПОЛНИМОЙ ПРОЦЕДУРЕ

При обращении к выполнимой процедуре из другой выполнимой процедуры или триггера используется следующий синтаксис:

```
EXECUTE PROCEDURE name [LIST_: param]
[RETURNING_VALUES [LIST_: param];
```

name - имя процедуры,

param - входной параметр или возвращаемое значение.

Символ «:» здесь является синтаксическим элементом.

Триггеры и хранимые процедуры 137

При обращении к выполнимой процедуре из ISQL используется следующий синтаксис.

```
EXECUTE PROCEDURE name [ ( ] [LIST_param] ) ];
```

name - имя процедуры,
param - входной параметр.

ОБРАЩЕНИЕ К ПРОЦЕДУРЕ ВЫБОРА

При обращении к процедуре выбора используется следующий синтаксис:

```
SELECT < col_list> from name (LIST_param)
[WHERE < search_condition>]
[ORDER BY < order_list>];
```

name - имя процедуры,
param - входной параметр процедуры,
< col_list> - список выбираемых выражений (должен базироваться на списке возвращаемых значений процедуры),
< search_condition>, < order_list> - условия выборки и способ упорядочения результатов.

В целом структура команды SELECT не зависит от того, каким образом выбираются данные (см. разд. 3.1). Синтаксически использование процедуры от таблицы или обзора отличается только тем, что после имени процедуры могут указываться ее параметры.

ДОСТУП К ХРАНИМЫМ ПРОЦЕДУРАМ

Доступ к хранимым процедурам, так же, как к таблицам и обзорам, регулируется механизмом предоставления прав доступа командами GRANT и REVOKE. Поскольку в своей работе хранимые процедуры могут обращаться к таблицам, обзорам и другим хранимым процедурам, самим процедурам также должны быть установлены права доступа. Изначально процедуры получают те же права, что и их создатель. Доступ к ним получает их создатель и пользователь SYSDBA или другой с аналогичными правами.

Глава 6

Расширенные возможности для работы с базой

6.1. Обзоры

Пользователи базы данных обычно должны обращаться к специфическому подмножеству данных, хранимых в базе. Кроме того, требования к данным отдельных пользователей или групп пользователей часто весьма противоречивы. Обзоры обеспечивают способ создать настраиваемую версию основных таблиц, которые отображают только те данные, которые интересуют данного пользователя или группу пользователей.

Как только обзор определен, с ним можно работать так, как если бы это была обычная таблица. Обзор может быть получен из одной или нескольких таблиц или других обзоров. Обзоры выглядят точно так же как обычные таблицы базы данных, но физически не хранятся в базе данных. В базе хранится только определение обзора, которое используется для фильтрации данных при выполнении запросов, ссылающихся на обзор.

Отметим, что создание обзора не создает копию данных, в то же время изменение данных через обзор, изменяет данные в основных таблицах, а замена данных в основных таблицах непосредственно отражается в обзоре. Обзор можно представлять как подвижное "окно", через которое видны фактические данные. Учитывая, что обзор все-таки не является таблицей, на операции с обзорами накладываются некоторые ограничения, которые будут рассмотрены ниже.

Обзор представляет собой виртуальную таблицу, которая создается на основе результатов выборки данных из базы командой SELECT. Он может быть создан:

- На основе подмножества столбцов отдельной таблицы.
- На основе подмножества строк отдельной таблицы.
- На основе комбинации подмножества строк и столбцов отдельной таблицы.

На основе комбинации подмножества строк и столбцов объединения нескольких таблиц.

Обзор может быть также создан на основе результатов работы хранимой процедуры.

Использование обзоров позволяет обеспечить решение ряда задач:

- Упрощенный доступ к данным. Обзоры дают возможность сформировать подмножество данных из одной или нескольких таблиц, которое можно использовать как основу для запросов без повторной выдачи команды SELECT. Кроме того, поскольку обзор является результатом выборки по команде SELECT, то SELECT от обзора позволяет фактически реализовать конструкцию типа SELECT <LIST_val> FROM ... (SELECT ...), которую непосредственно нельзя реализовать.
- Настраиваемый доступ к данным. Обзоры обеспечивают способ приспособить базу данных к требованиям разных пользователей с различными привычками и интересами. Обзоры позволяют выделить только те данные, которые интересуют конкретного пользователя.
- Независимость приложений от организации хранения данных. Обзоры изолируют пользователей от изменений в структуре основной базы данных (при изменении базы несложно сохранить обзор в неизменном виде).
- Защита данных. Обзоры обеспечивают защиту данных, ограничивая доступ к отдельным элементам данных.

Синтаксис создания обзора

```
CREATE VIEW name [( view_col [, view_col ...])]
AS <select> [WITH CHECK OPTION];
```

Таблица 6.1. *Синтаксические конструкции команды CREATE VIEW*

Конструкция	*Описание*
name	Имя обзора. Имя должно быть уникальным в перечне имен обзоров, таблиц и хранимых процедур

Конструкция	*Описание*
view_col	Имя столбца обзора. Имя должно быть уникальным в перечне имен столбцов обзора. Имя обязательно, если обзор включает столбцы-выражения. Если не указано, используется имя соответствующего столбца таблицы из SELECT. Имена столбцов обзора соответствуют столбцам выборки SELECT
select	Задает условия выборки данных посредством команды SELECT. Может использоваться полный синтаксис SELECT за исключением конструкции ORDER BY (см. разд. 3.1)
WITH CHECK OPTION	Предотвращает операции INSERT или UPDATE в обзоре, если они нарушают условие поиска в конструкции WHERE команды SELECT

Пользователь, создавший обзор является его владельцем и имеет на него все права, включая право передачи прав другим пользователям, триггерам и процедурам. Пользователь может получить права на обзор без получения доступа к исходным таблицам.

Типы обзоров

Обзоры могут быть обновляемыми (updatable) или только для чтения (read-only).

Чтобы обзор был обновляемым, необходимо:

- чтобы он представлял собой подмножество одной таблицы или обновляемого обзора;
- чтобы все столбцы таблицы, не вошедшие в обзор, допускали значение NULL;

чтобы команда SELECT, на которой основан обзор, не содержала подзапросов, конструкций DISTINCT, HAVING, агрегатных функций, присоединенных таблиц, пользовательских функций или хранимых процедур.

Обзоры только для чтения также могут обновляться, но на *основе использования триггеров*. В этом случае все действия по обновлению данных в таблицах выполняются соответствующими триггерами обзора.

Для обеспечения возможности обновления данных (команды UPDATE и INSERT) пользователь должен получить соответствующие права, кроме того, для создания обновляемого обзора необходимо, чтобы его создатель имел все права на таблицы, используемые обзором. Для обзора только для чтения достаточно иметь права на SELECT.

Пример 6.1

```
create view RUBRICS as
select UNIKEY, BOOKNM from tbook
where (matherkey=0)
```

Таблица 6.2. *Перечень рубрик - обзор RUBRICS*

UNIKEY	*BOOKNM*
2	Программирование
3	Учебники
4	Математика
5	Беллетристика

Теперь попробуем выполнить обновление (обзор обновляемый).

Пример 6.2

```
update RUBRICS
set BOOKNM= 'Программирование и алг. языки'
where unikey=2
```

Изменение выполнено

Пример 6.3

```
create view NORUBRICS as
select UNIKEY, BOOKNM, B1 from tbook a, tbook_author b
where a.matherkey>0 and a.unikey=b.bookkey
```

Таблица 6.3. *Перечень книг с указанием авторов - обзор RUBRICS*

UNIKEY	*BOOKNM*	*B1*
58	Макрокоманды MS Word	Культин Н.Б.
59	Введение в технологию ATM	Буассо Марк
60	Введение в технологию ATM	Деманж Мишель
61	Введение в технологию ATM	Мюнье Жан-Мари
62	С и С++ Справочник	Луис Дерк
63	Borland-Технологии. SQL-Link Inter-Base, Paradox for Windows, Delphi	Дунаев Сергей

UNIKEY	BOOKNM	B1
64	Введение в C++ Builder	Елманова Н.З.
65	Введение в C++ Builder	Кошель С П.
66	Язык C++	Подбельский Вадим Валериевич
67	Word 6 for Windows	Хаселир Райнер Г.
68	Word 6 for Windows	Фаненштих Клаус
69	Математические вопросы динамики вязкой несжимаемой жидкости	Ладыжинская Ольга Александровна
70	Тесты. Сборник 11 класс. Варианты и ответы государственного тестирования. Пособие для подготовки к тестированию	без авторов
71	Справочник по правописанию и литературной правке	Розенталь Д.Э.
72	The history of England. Absolute Monarchy	Бурова И.И.
73	Кровь нерожденных	Дашкова Полина
74	Тайна	Хмелевская Иоанна

Теперь попробуем выполнить обновление (обзор только на чтение, используются две таблицы, к тому же столбец B1 - вычисляемый).

```
update NORUBRICS
set BOOKNM='Тайна!!!'
where unikey=74
```

Результат - The object of the insert, delete or update statement is a view for which the requested operation is not permitted. Cannot update read-only viewNORUBRICS.

Обновление данных в обзоре с конструкцией WITH CHECKOPTION

Если обзор создан с опцией проверки **(WITH CHECK OPTION),** то при попытке обновления или вставки данных проверяется, удовлетворяют ли новые данные условиям выборки, заданным конструкцией WHERE. Данные будут записаны только при выполнении этих условий. Опция **WITH** CHECK OPTION применима только для обновляемых обзоров.

Расширенные возможности для работы с базой 143

Значения могут быть вставлены через обзор только для тех столбцов, которые входят в обзор. Для столбцов таблицы, не вошедших в обзор, InterBase устанавливает значения NULL.

Изменение обзора

Нельзя непосредственно изменить обзор. Необходимо сначала выполнить его удаление (DROP VIEW), а затем создать его вновь командой CREATE VIEW с требуемыми характеристиками.

Удаление обзоров

Удаление обзора означает удаление из базы его описания, но не самих данных. Оно разрешено пользователю, создавшему обзор, при выполнении ряда условий, а именно: обзор не используется другим обзором, хранимой процедурой или ограничением целостности (CHECK-конструкцией). В этом случае необходимо предварительно удалить все объекты, использующие данный обзор.

Синтаксис:

```
DROP VIEW name;
```

6.2. Работа с BLOB

Рассмотрим подробнее работу с данными BLOB (большой двоичный объект), подтипами BLOB, особенностями чтения и записи BLOB, обращениям к BLOB с помощью SQL, DSQL и вызовами API.

Что представляет собой BLOB?

BLOB предназначен для хранения данных произвольного формата переменной длины и, как правило, значительного размера.

BLOB можно использовать для хранения объектов разной природы и назначения, включая:

- Растровые изображения.
- Векторные рисунки.
- Звуки, видео и другую информацию мультимедиа.
- Текст и данные, включая документы большого объема.

BLOB представляет собой тип данных с динамически изменяемым размером, для этого типа не указываются ни размеры, ни способ кодирования. Внутри таблиц вместо данных BLOB записываются их уникальные дескрипторы (указатели) фиксированного размера на место их фактического хранения. Благодаря этому доступ к дескрипторам данным BLOB в InterBase осуществляется так же, как к данным, хранимым в других

форматах. В отличие от ряда других систем, хранящих данные BLOB типа во внешних файлах, InterBase хранит данные BLOB внутри базы. Для каждого BLOB имеется уникальный дескриптор (указатель) в соответствующей таблице, описывающий местоположение BLOB в базе. Поддержка хранения BLOB в пределах базы данных, позволяет гарантировать их от случайных изменений и унифицировать организацию управления хранением и доступом к данным.

Сочетание непосредственного управления BLOB данными в базе данных и поддержка разнообразных типов делает механизм BLOB InterBase пригодным для поддержки приложений, интенсивно работающих с мультимедиа или хранения больших объемов текстов. Например, для приложений типа магазина, хранящих данные как о продажах, так и образцах типа фотографий, файлов видеозаписи вместе с возможностью обработки заказов и продаж. Другим примером может служить библиотечная система, хранящая данные о книгах, работе абонемента и т.п. вместе с рефератами по всем единицам хранения.

Хранение данных BLOB

BLOB - тип данных InterBase, обеспечивающий хранение данных, который представляет различные объекты вроде растровых изображений, звука, видео и текста. Прежде чем сохранить эти элементы в базе данных, они создаются как файлы определенной структуры, например:

- TIFF, PICT, BMP, WMF, GEM, TARGA или другие растровые или векторно-графические файлы.
- MIDI или WAV звуковые файлы.
- Интерактивные аудио-видео файлы AVI (Audio Video Interleaved) или видео файлы формата QuickTime.
- ASCII, MIF, DOC, RTF, WPx или другие текстовые файлы.
- Файлы CAD.

Затем эти файлы программно загружаются в базу данных. Доступ к данным также осуществляется программно. По сути, здесь можно провести следующую аналогию. Таблица с BLOB - директория, дескриптор BLOB - имя файла, данные BLOB - файл. В соответствии с этим и осуществляется доступ к объектам BLOB.

Подтипы BLOB

Хотя все виды хранимых в BLOB данных обрабатывается однотипно, но как для внутренних целей, так и для внешнего использования полезно хранить информацию о типе данных, содержащихся в BLOB. Поскольку имеется много естественных типов данных, которые можно определить как BLOB, InterBase позволяет определить подтип BLOB, обеспечивает

семь стандартных подтипов, с помощью которых можно описывать BLOB, а также, практически неограниченное количество пользовательских подтипов.

Таблица 6.4. *Подтипы BLOB*

Подтип	***Описание***
0	Неструктурированный тип, обычно применимый к двоичным данным или данным неопределенного типа
1	Текст
2	Двоичное представление языка (BLR)
3	Список контроля доступа (Access control list)
4	(Резерв для будущего использования)
5	Закодированное описание метаданных таблиц
6	Описание неуспешных транзакции, работающих с несколькими базами данных

Пользовательский подтип можно определить, как отрицательное число в интервале от -1 до -32 678.

Положительные целые числа зарезервированы для стандартных подтипов InterBase, для части из них предусмотрена и своя обработка.

Столбцы BLOB определяются стандартным образов в командах CREATE TABLE, ALTER TABLE. Например, следующая команда определяет три столбца BLOB:

BLOB1 с подтипом 0 (принят по умолчанию), BLOB2 с подтипом 1 (текст) и BLOB3 с пользовательским подтипом **-1.**

Пример 6.4

```
CREATE TABLE TABLEBLOB
 (
BLOB1 BLOB,
BLOB2 BLOB SUB_TYPE 1,
BLOB3 BLOB SUB_TYPE -1
);
```

Чтобы определить и заданный по умолчанию размер сегмента, и подтип при создании столбца BLOB, используется опция SEGMENT SIZE, записываемая после опции SUB_TYPE. В приведенном ниже примере описывается создание в нашей тестовой базе описание таблицы книг, со-

держащей столбец REFERAT для хранения текста реферата книги, представляющий собой BLOB типа 0 (произвольный объект) с 80-байтовым сегментом.

Пример 6.5

```
CREATE TABLE TBOOK (
    UNIKEY PRMKEY,
    MATHERKEY INTEGER,
    BOOKNM VARCHAR(250),
    REFERAT BLOB sub_type 0 segment size 80,
    NUM_ALL SMALLINT DEFAULT 0 NOT NULL,
    NUM_PRESENCE SMALLINT DEFAULT 0 NOT NULL);
```

Единственное требование, которое InterBase предъявляет для определяемых пользователем подтипов - совместимость при преобразовании BLOB от одного подтипа к другому.

Использование памяти для BLOB

Поскольку данные BLOB - обычно большие и переменного размера объекты двоичных или текстовых данных, InterBase хранит их, используя метод сегментации. Использовать дисковое пространства для хранения каждого BLOB в одном непрерывном участке из-за возможных изменений длин объектов было бы неэффективно, поскольку их перезапись потребовала бы либо перемещения всех BLOB, либо привела бы к возникновению большого количества «дыр», устранение которых также требует периодической реорганизации базы. Вместо этого InterBase хранит каждый BLOB в сегментах, которые индексируются дескриптором, генерируемым InterBase, когда создается BLOB. Этот дескриптор называется идентификатором BLOB (BLOB ID) и представляет собой учетверенное слово (64 разряда), содержащее уникальную комбинацию идентификатора таблицы и идентификатора BLOB.

BLOB ID для каждого BLOB хранится в соответствующем поле записи таблицы. BLOB ID указывает на первый сегмент BLOB или на страницу указателей, каждый из которых указывает на сегмент одного или нескольких полей BLOB. BLOB ID можно получить командой SELECT, которая определяет BLOB как адресат.

Работа с BLOB осуществляется на базовом языке. Сам доступ реализуется, как правило, одним из трех способов.

С использованием специально включаемого в программу SQL текста. В этом случае программа перед ее компиляцией обрабатывается специальным препроцессором. Для InterBase таковым является утилита GPRE. В результате ее работы этот текст транслируется в последовательность вызовов функций API. Поскольку аналогичные методы используются

Расширенные возможности для работы с базой 147

и другими СУБД, то это создает возможность с минимальными потерями переходить от работы с одной базы на другую.

Второй способ предполагает прямой вызов функций API. В этом случае программа оказывается явно привязанной к платформе InterBase, зато не требует предварительной препроцессорной обработки. Этот метод наиболее выгоден при написании стандартных функций, например, при создании библиотеки UDF (User Defined Functions).

Третий способ представляет собой использования средств доступа высокого уровня. В самом деле, поскольку работа с объектами BLOB первыми двумя способами достаточно трудоемка и при этом по своей сути стандартна, то было бы странно, если бы не было подобных высокоуровневых средств доступа к ним. В частности в системах C++ Builder и Delphi имеются специальные объекты для работы с BLOB.

Рассмотрим на примерах работу с BLOB различными методами.

В этом разделе будем использовать первый методами.

Использование API будет проиллюстрировано в разделе о создании UDF.

Использование высокоуровневых методов рассмотрим в специальном разделе. Для большинства приложений именно они являются предпочтительными, так как позволяют абстрагироваться от деталей процедур чтения и записи и сосредоточиться только на содержательной части задачи, сохраняя при этом высокую эффективность обработки данных.

Выборка BLOB ID с использованием внедренного SQL показана в следующем примере (тексты этих примеров должны, естественно, перед их выполнением быть обработаны препроцессором). Предварительно создается курсор BLOB, представляющий отдельную строку выборки, содержащей объект BLOB (к курсорам вернемся несколько позже, а пока этот пример следует рассматривать, как иллюстрацию синтаксиса).

Пример 6.6

```
EXEC SQL
DECLARE BLOBDESC CURSOR FOR
SELECT REFERAT
FROM TBOOK
WHERE UNIKEY = 23;
```

Столбцы BLOB определяются так же, как обычные при создании таблиц (см. пример 6.5).

Диаграмма на рис. 6.1 показывает связь между столбцом BLOB, содержащим BLOB ID, и данными, на которые он указывает.

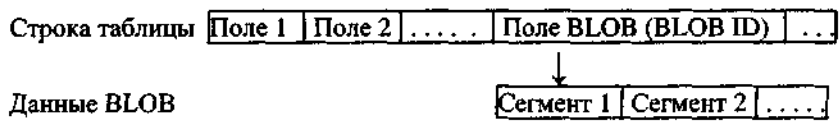


Рис. 6.1. Связь между столбцом BLOB и данными BLOB.

Вместо хранения данных BLOB непосредственно в таблице, InterBase хранит в ней только идентификатор. В базе данных ID указывает на первый сегмент данных BLOB, который хранится в базе данных в другом месте - в списке сегментов. Когда приложение создает BLOB в базе данных, оно записывает его содержимое посегментно. Точно так же приложение читает BLOB. Поскольку большинство данных BLOB - большие объекты, доступ к BLOB в большинстве случаев реализуется циклами в прикладном коде.

Длина сегмента BLOB

При определении BLOB в таблице в команде определения BLOB задается ожидаемый размер сегментов BLOB, которые должны быть записаны в столбец. Длина сегмента, определяемая для столбца сегмента, задает максимальное число байтов, которые приложение, как ожидается, запишет или будет читать из любого BLOB в столбце. Заданная по умолчанию длина сегмента - 80. Например, следующее объявление столбца создает BLOB с длиной сегмента 120:

Пример 6.7

```
EXEC SQL
CREATE TABLE TABLEBLOB
(
BLOB1 BLOB SEGMENT SIZE 120;
);
```

InterBase использует установку длины сегмента, чтобы определить размер внутреннего буфера, в который записывает данные сегмента. Обычно не следует пытаться записывать сегменты большей длины, чем специфицировано в таблице; выполнение таких команд может привести к переполнению буфера и возможному искажению памяти.

Указание длины сегмента *n* гарантирует, что не более чем *n* байтов будет прочитано или записано за одну операцию с BLOB.

С некоторыми типами операций, например SELECT, INSERT или UPDATE, можно читать или записывать сегменты BLOB переменной длины.

В следующем примере команды INSERT CURSOR указывается длина сегмента в переменной базового языка *segment_length.*

Пример 6.8

```
EXEC SQL
INSERT CURSOR BLOBINS VALUES (:write_segment_buffer
INDICATOR :segment_length);
```

Отмена длины сегмента

Можно отменять установку длины сегмента включением опции MAXIMUM_SEGMENT в инструкции DECLARE CURSOR. Так, следующее объявление курсора BLOB INSERT отменяет длину сегмента, которая была определена для поля BLOB1, увеличивая ее до 1024:

Пример 6.9

```
EXEC SQL
DECLARE BLOBINS CURSOR FOR INSERT BLOB BLOB1 INTO TABLEBLOB
MAXIMUM_SEGMENT 1024;
```

Примечание. Отмена установки длины сегмента затрагивает только размер сегмента для курсора, но не для столбца таблицы базы данных или других курсоров. Другие курсоры, использующие тот же самый столбец BLOB, поддерживают размер сегмента, который был задан при определении столбца, или могут определить собственные длины.

Установка длины не затрагивает системное представление InterBase. При выборе длины сегмента следует руководствоваться удобством для конкретного приложения. Максимальная возможная длина сегмента - 65 535 байт (64K).

ДОСТУП К BLOB С ИСПОЛЬЗОВАНИЕМ SQL (ИЗ ПРОГРАММЫ НА ЯЗЫКЕ С)

InterBase поддерживает команды SELECT, INSERT, UPDATE и DELETE для BLOB. Ниже приводятся примеры соответствующих программ, иллюстрирующих применение стандартного SQL для работы с BLOB. (Текст должен быть перед выполнением обработан утилитой Gpre).

Выборка BLOB

Следующая программа выбирает данные BLOB из столбца REFERAT таблицы TBOOK. Для реализации выборки нужно выполнить последовательно ряд действий:

- Объявить переменные базового языка для записи BLOB ID, данных сегментов BLOB и длины сегмента данных.
- Объявить курсор таблицы для выборки требуемого столбца BLOB.
- Объявить курсор для чтения BLOB, необходимый для чтения его сегментов.

- Открыть курсор таблицы и выбрать строку данных, содержащих BLOB.
- Открыть курсор чтения BLOB и выбрать первый сегмент данных.
- Выбрать в цикле остающиеся сегменты.
- Закрыть курсор чтения BLOB.
- Закрывает курсор для таблицы.

Пример 6.10

1. Объявляются переменные базового языка для записи BLOB ID, данных сегментов BLOB и длины сегмента данных:

```
EXEC SQL
BEGIN DECLARE SECTION;
BASED ON TBOOK.REFERAT blob_id;
BASED ON TBOOK.REFERAT.SEGMENT blob_segment_buf;
BASED ON TBOOK.UNIKEY key;
unsigned short blob_seg_len;
EXEC SQL
END DECLARE SECTION;
```

Конструкция BASED ON ... SEGMENT объявляет переменную базового языка blob_segment_buf, которая должна иметь размер, достаточный для размещения сегмента BLOB во время выполнения команды FETCH.

2. Объявляется курсор таблицы для выборки требуемого столбца BLOB, в данном примере - столбец REFERAT:

```
EXEC SQL
DECLARE TABCURSOR CURSOR FOR
SELECT UNIKEY, REFERAT
FROM TBOOK
WHERE REFERAT = 123;
```

3. Объявляется курсор для чтения BLOB. Курсор для чтения BLOB - специальный тип курсора, используемый для чтения сегментов BLOB:

```
EXEC SQL
DECLARE BLOBCURSOR CURSOR FOR
READ BLOB REFERAT
FROM TBOOK;
```

Длина сегмента столбца BLOB REFERAT определена как 80, курсор BLOB BLOBCURSOR читает максимум 60 байт одновременно.

Расширенные возможности для работы с базой 151

Чтобы переопределить длину сегмента, указанную в схеме базы данных для REFERAT, используется опция MAXIMUMSEGMENT. Например, следующий код ограничивает каждую операцию чтения BLOB максимумом в 60 байт, и SQLCODE устанавливается в 101, чтобы указать, когда прочитана только часть сегмента (признак конца данных):

```
EXEC SQL
DECLARE BLOBCURSOR CURSOR FOR
READ BLOB REFERAT
FROM TBOOK
MAXIMUM_SEGMENT 60;
```

Независимо от того, какая длина сегмента установлена, за одну операцию чтения считывается ровно один сегмент.

4. Открывается курсор таблицы и выбирается строка данных, содержащая BLOB:

```
EXEC SQL
OPEN TABCURSOR;
EXEC SQL
FETCH TABCURSOR INTO :key, :blob_id;
```

Команда FETCH выбирает столбцы UNIKEY и REFERAT в host-переменные key и blob_id соответственно.

5. Открывается курсор чтения BLOB (путем использования BLOB ID), находящийся в переменной blob_id, и выбирается первый сегмент данных BLOB:

```
EXEC SQL
OPEN BLOBCURSOR USING :blob_id;
EXEC SQL
FETCH BLOBCURSOR INTO :blob_segment_buf:blob_seg_len;
```

Когда операция FETCH завершается, blob_segment_buf содержит первый сегмент данных BLOB, blob_seg_len содержит длину сегмента (число байтов, скопированных в blob_segment_buf).

6. Выбираются остающиеся сегменты в цикле на базовом языке, используя объявленные ранее переменные. После каждой выборки проверяется SQLCODE. Код 100 указывает, что все данные BLOB были выбраны. Код 101 указывает, что в сегменте BLOB еще остались данные. В приведенном примере полученные данные печатаются. Здесь их просто некуда деть, в реальных задачах данные либо сохраняются на диске, либо помещаются в какой-либо объект для последующей визуализации:

```
while (SQLCODE != 100 || SQLCODE == 101)
{
printf("%*.*s", blob_seg_len, blob_seg_len,
blob_segment_buf);
EXEC SQL
FETCH BC INTO :blob_segment_buf:blob_seg_len;
}
```

InterBase устанавливает код ошибки 101, когда длина буфера сегмента меньше, чем его специфицированная длина.

Например, как это происходит в нашем случае, если длина буфера сегмента 60, а длина специфицированного сегмента 80, то первый FETCH устанавливает код ошибки 101, указывающий, что в сегменте еще остаются данные. Второй FETCH читает оставшиеся 20 байт данных и устанавливает SQLCODE 0, указывающий, что следующий сегмент готов к чтению, или 100, если это был последний сегмент в BLOB.

7. Закрывает курсор чтения BLOB:

```
EXEC SQL

CLOSE BLOBCURSOR;
```

8. Закрывает курсор для таблицы:

```
EXEC SQL
CLOSE TABCURSOR;
```

Вставка данных в BLOB

Следующая программа полностью аналогична предыдущей, но предназначена для вставки данных в BLOB-столбец REFERAT таблицы TBOOK. Для реализации вставки, также как и выборки нужно выполнить последовательно ряд действий:

- Объявить переменные базового языка для BLOB ID, данных сегмента BLOB и длины сегмента BLOB.
- Объявить BLOB-курсор для вставки.
- Открыть BLOB-курсор для вставки и задать host-переменную для размещения BLOB ID.
- Записать данные сегмента в буфере сегментов.
- Закрыть BLOB-курсор для вставки.
- Выполнить вставку новой строки, содержащей BLOB, в таблицу.
- Зафиксировать изменения в базе.

Пример 6.11

1. Объявляются переменные базового языка для BLOB ID, данных сегмента **BLOB и длины сегмента BLOB:**

```
EXEC SQL
BEGIN DECLARE SECTION;
BASED ON TBOOK.REFERAT blob_id;
BASED ON TBOOK.REFERAT.SEGMENT blob_segment_buf;
BASED ON TBOOK.UNIKEY key;
unsigned short blob_seg_len;
EXEC SQL
END DECLARE SECTION;
```

Конструкция BASED ON ... SEGMENT объявляет переменную базового языка *blob_segment_buf,* которая должна иметь размер, достаточный для размещения сегмента BLOB во время выполнения команды FETCH.

2. Объявляется BLOB-курсор для вставки:

```
EXEC SQL
DECLARE BLOBCURSOR CURSOR FOR
INSERT INTO TBOOK;
```

Курсор для работы с таблицей, как в случае выборки, здесь не требуется. Будем считать, что все данные, загружаемой в базу строки, включая первичный ключ, нам известны.

3. Открывается BLOB-курсор для вставки и задается host-переменная для размещения BLOB ID:

```
EXEC SQL
OPEN BLOBCURSOR INTO :blob_id;
```

4. Записываются данные сегмента в буфере сегментов *blob_segment_buf,* вычисляется длина сегмента данных и используется команда INSERT CURSOR для записи сегмента. Эти действия повторяются в цикле, пока не будут записаны все сегменты BLOB:

```
char **s_referat; // Массив указателей на строки реферата
int n_strings;    // Количество строк в реферате
. . . .

for(int i=0;i<n_strings;i++)
{
 Sprintf (blob_segment_buf, s_referat[i]);
 blob_segment_len = strlen(blob_segment_buf);
```

```
 EXEC SQL
 INSERT CURSOR BLOBCURSOR
  VALUES(:blob_segment_buf:blob_segment_len);
}
```

5. Закрывается BLOB-курсор для вставки:

```
EXEC SQL
CLOSE BLOBCURSOR;
```

6. Команда INSERT используется для вставки новых строк, содержащих BLOB, в таблицу **TBOOK:**

```
EXEC SQL
INSERT INTO TBOOK (UNIKEY, MATHERKEY, BOOKNM, REFERAT)
VALUES (188, 44, 'Неведомая книга', :blob_id);
```

Данный пример ввода непосредственно в таком виде, конечно, плох, поскольку столбцы UNIKEY и MATHERKEY являются ключевыми и их значения не могут быть произвольными. Они должны быть, вообще говоря, определены в результате выборки и обработки данных из базы.

7. Фиксируются изменения в базе:

```
EXEC SQL
COMMIT;
```

Обновление данных BLOB

Непосредственно модифицировать BLOB нельзя. Необходимо создать новый BLOB и либо считать старые данные BLOB в буфер, где их можете редактировать или изменять, а затем записать измененные данные в новый BLOB, либо создать новое значения без явного чтения старого BLOB. Для модификации нужно выполнить последовательно ряд действий:

- Объявить BLOB-курсор для вставки.
- Открыть BLOB-курсор для вставки и задать host-переменную для размещения BLOB ID.
- Считать сегменты данных старого BLOB, модифицировать их и записать в базу.
- Закрывает BLOB-курсор для вставки.
- Выполнить команду UPDATE для замены BLOB.

Расширенные возможности для работы с базой 155

Пример 6.12

1. Объявляется BLOB-курсор для вставки:

```
EXEC SQL
DECLARE BLOBCURSOR CURSOR FOR INSERT BLOB REFERAT INTO
TBOOK;
```

2. Открывается BLOB-курсор для вставки и задается host-переменная для размещения BLOB ID:

```
EXEC SQL
OPEN BLOBCURSOR INTO :blob_id;
```

Здесь предполагается, что курсор для таблицы уже открыт и мы, следовательно, настроены на работу с определенной строкой таблицы (см. пример 6.10).

3. Записывается сегмент данных BLOB в буфер сегментов *blob_segment_buf,* вычисляется длина сегмента данных, выполняются действия по модификации данных, и используется команда INSERT CURSOR для записи сегмента:

```
EXEC SQL
INSERT CURSOR BLOBCURSOR VALUES
(:blob_segment_buf:blob_segment_len);
```

Эти действия повторяются в цикле, пока не будут записаны все сегменты BLOB.

4. Закрывает BLOB-курсор для вставки:

```
EXEC SQL
CLOSE BLOBCURSOR;
```

5. Когда процесс создания нового BLOB завершен, выполняется команда UPDATE, чтобы заменить старый BLOB в таблице новым, как показано ниже:

```
EXEC SQL UPDATE TBOOK
SET
REFERAT = :blob_id;
WHERE CURRENT OF TABCURSOR;
```

Курсор таблицы TABCURSOR указывает на строку, установленную при объявлении курсора, а затем выбирает ее для обновления.

УдалениеданныхBLOB

Существует два метода для удаления BLOB. Во-первых, можно удалить строку, содержащую BLOB. Во-вторых, можно модифицировать строку и установить столбец BLOB в NULL или в BLOB ID другого BLOB (например, нового BLOB, созданного для модификации данных существующего).

Следующая команда удаляет текущие данные BLOB в столбце REFERAT таблицы TBOOK, устанавливая его в NULL:

Пример 6.13

```
EXEC SQL
UPDATE TBOOK
SET
REFERAT = NULL;
WHERE CURRENT OF TABCURSOR;
```

То же самое можно сделать и используя интерактивный SQL, поскольку содержимое BLOB здесь не используется.

Пример 6.14

```
UPDATE TBOOK
SET
REFERAT = NULL;
WHEREUNIKEY=123;
```

Удаление целиком строки с **BLOB полностью аналогично удалению** любой другой строки таблицы.

```
DELETE FROM TBOOK
WHEREUNIKEY=123;
```

Данные BLOB не удаляются немедленно при выполнении команды удаления. Фактическое удаление происходит, когда InterBase выполняет очистку версии. Подробнее работа с версиями описана в гл. 9. Следующий фрагмент кода иллюстрирует, как освободить память после удаления BLOB:

Пример 6.15

```
EXEC SQL
UPDATE TABLE SET BLOB_COLUMN = NULL WHERE ROW = :myrow;
EXEC SQL
COMMIT;
. . . .
```

```
/* Выполняем текущие действия */
. . . .
/* Ждем события, подтверждающего завершение всех активных
на момент старта удаления транзакций */
. . . .
/* Запускаем чистку базы */
. . . .
```

Если этого не делать, то чистка все равно будет произведена, но в то время, когда InterBase самостоятельно сочтет это нужным, то есть тогда, когда InterBase выполняет сборку "мусора" от старых версий записей. Подробнее механизм чистки описан в гл. 9 о работе с транзакциями.

Доступ к данным BLOB через вызовы API

В дополнение к доступу к данным BLOB, использующим SQL, InterBase API обеспечивает подпрограммы для доступа к данным BLOB. Следующие вызовы API обеспечивают доступ и управления данными BLOB.

Таблица 6.5. *Функции API для работы с BLOB*

Функция	*Описание*
isc_blob_default_desc()	Загружает структуру дескриптора BLOB заданной по умолчанию информацией о BLOB
isc_blob_gen_bpb()	Генерирует буфер параметров BLOB (BPB) исходного и целевого дескрипторов BLOB, чтобы обеспечить динамический доступ к подтипу BLOB и используемой кодовой таблице (набору) символов
isc_blob_info()	Возвращает информацию об открытом BLOB
isc_blob_lookup_desc()	Просматривает и записывает в дескриптор BLOB подтип, набор символов и размер сегмента BLOB
isc_blob_set_desc()	Устанавливает в полях дескриптора BLOB значения, указанные в параметрах isc_blob_set_desc()
isc_cancel_blob()	Отказывается от BLOB и освобождает оперативную память
isc_close_blob()	Закрывает открытый BLOB
isc_create_blob2()	Создает контекст для сохранения BLOB, открывает BLOB для записи и определяет фильтр (необязательная опция), который нужно использовать, чтобы транслировать данные BLOB из одного подтипа в другой

Функция	*Описание*
isc_get_segment()	Читает сегмент из открытого BLOB
isc_open_blob2()	Открывает существующий BLOB для выборки и необязательной фильтрации
isc_put_segment()	Записывает сегмент BLOB

Примеры использования API приведены в разделе, описывающем работу с UDF (фильтры BLOB).

ДОСТУП К BLOB ИЗ ПРОГРАММ НА C++ BUILDER И DELPHI

Программирование доступа к BLOB из прикладных программ, используя механизм BLOB-курсоров с предварительной обработкой препроцессором GPRE, требует значительных усилий и чревато большим количеством ошибок. Для прикладных программистов одним из лучших решений при реализации приложений, работающих с базами данных в среде Windows, является выбор в качестве средства разработки программ систем C++ Builder и Delphi фирмы Inprise (Borland).

Рассмотрим порядок работы с BLOB в C++ Builder.

В этом случае от прикладного программиста не требуется знание механизма доступа к BLOB. Для чтения данных необходимо установить связь с таблицей, из которой выбираются данные.

При работе с любыми базами данных, включая и локальные файлы, необходимо указать источник данных и описать свойства этих данных и порядок их обработки. Для такого рода работ в C++ Builder (Delphi) предусмотрены специальные объекты. Прежде всего, это такие объекты, как TTaЫe (таблица) или TQuery (запрос).

При работе с ними необходимо описать некоторые из их свойств. Прежде всего, нужно указать откуда берутся данные. Для этого используется свойство объекта DatabaseName, задающее имя базы данных. Далее нужно указать либо имя таблицы базы данных при работе с таблицами в свойстве TableName объекта TTaЫe, либо записать команду SQL (в данном случае Select) в свойстве SQL объекта TQuery.

Указанные объекты используют механизм BDE (Borland Database Engine), позволяющий одинаково работать с различными базами данных. Поскольку BDE может работать с разными СУБД, то предварительно нужно провести настройку BDE, это достаточно простая и к тому же разовая процедура. О ней мы поговорим несколько позже. Если используются последние версии C++ Builder, то в них есть средства прямого доступа к InterBase. В этом случае можно воспользоваться объектами ТШТаЫе или TIBQuery. Чтобы не привязываться к конкретным версиям продуктов, остановимся на доступе к данным через BDE.

Визуализация и корректировка данных BLOB

Прежде всего, рассмотрим выдачу данных BLOB в визуальный объект. При этом допустима и корректировка этого объекта, которая отражается в базе данных.

Пример 6.16

Итак, создаем форму и помещаем на нее 4 объекта:

- Table 1 - TTable, в котором описываем привязку к базе данных. Пусть в BDE имя нашей тестовой базы Test.gdb - TESTLIBR, тогда свойство DatabaseName будет TESTLIBR, свойство TableName (имя таблицы) - TBOOK.
- DataSource1 - TTDataSource - объект необходимый для связи таблицы с визуальными объектами. Его свойство DataSet установим в Table 1, связав TDataSourcel с нужной нам таблицей.
- DBGridl - TDBGrid - объект для просмотра и редактирования данных в табличной форме. Свойство объекта DataSource установим в DataSourcel. Теперь мы через DataSourcel связали наш визуальный объект с таблицей TBOOK нашей базы.
- DBMemol - TDBMemo - объект для просмотра редактирования текстов, хранящихся в базе данных. Свойство объекта DataSource установим в DataSourcel. Теперь через DataSourcel мы связываем DBMemol с таблицей TBOOK, а, указав в свойстве DataField значение REFERAT, указали, что в нем будет отображаться содержимое поля REFERAT, представляющее собой текстовый BLOB.

Установим теперь свойство Active объекта Table 1 в *true* (это эквивалентно открытию таблицы).

Результат работы представлен на рис. 6.2.

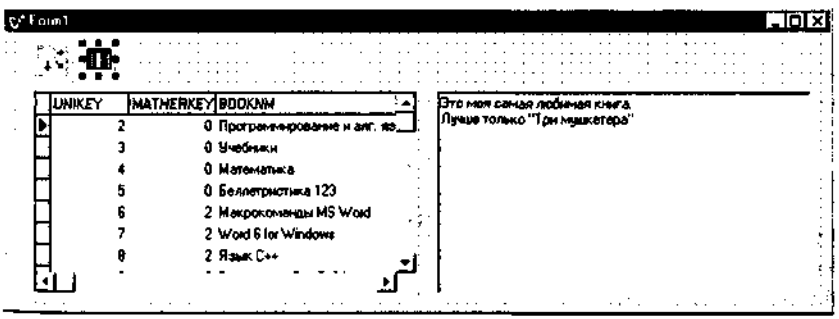


Рис. 6.2. Представление данных базы с BLOB на экране.

На этом можно и закончить нашу работу. При перемещении по таблицы в окне DBMemol будет высвечиваться содержимое очередного реферата. То же самое можно проделать и с BLOB полями, хранящими графические данные, только вместо объекта TDBMemo нужно будет поместить объект TDBImage.

Как видно из сказанного, для этого не понадобилось ни одной строки кода, все сделано стандартными средствами (последние, конечно же, используют соответствующие средства API, но для прикладного программирования это уже не имеет значения).

То же самое можно сделать и использую вместо объекта TTable объекта TQuery.

В этом случае вместо указания имени таблицы нужно указать порядок выборки данных. Для этого нужно указать соответствующую команду SELECT. Для ее задания используется свойство SQL объекта TQuery. В нашем случае это может быть конструкция вида

```
SELECT * FROM TBOOK
```

Для того чтобы автоматически генерировались команды по обновлению данных, свойство RequestLive должно быть установлено в ***true.*** Естественно не по любому запросу можно просто сгенерировать необходимые команды изменений, например, если выбираются данные из пяти таблиц с вычисляемыми значениями, то такая генерация просто невозможна, поэтому свойство RequestLive=true применимо только к выборкам из одной таблицы. Подробное описание этих ограничений можно найти в документации по C++Builder и Delphi, но это выходит за рамки данной книги.

Если выбираемые данные не нужно корректировать, то ограничений на вид команды SELECT нет, кроме того процедуры корректировки данных по результатам выборки можно описать и явно, предусматривая соответствующие команды INSERT, UPDATE и DELETE, тогда также команда SELECT может быть любой, например:

```
select * FROM PBOOKAUTHOR
```

Здесь для выборки используется вообще не таблица, а процедура PBOOKAUTHOR, формирующая список авторов в дополнения к сведениям о книгах. Но о процедурах речь еще впереди.

Запись данных BLOB в файл

Запись данных из BLOB в файл с использованием объекта TTable проиллюстрируем следующим примером. Проделаем те же действия по настройке таблиц (запроса), что и в предыдущем примере. Тогда запись в файл будет реализовываться одной строкой кода.

Пример 6.17

```
. . .              // Находим нужную строку
TBLOBField * tt=
```

Расширенные возможности для работы с базой 161

```
 dynamic_cast<TBLOBField *>
(Table1->FieldByName("REFERAT"));
// Настраиваемся на работу с BLOB
 tt->SaveToFile("AAA.AAA");
// Записываем данные BLOB в файл
```

При использовании для записи данных из **BLOB** объекта TQuery код будет совершенно аналогичным:

```
. . .                        // Находим нужную строку
TBLOBField * tt=
 dynamic_cast<TBLOBField *>
(Query1->FieldByName("REFERAT"));
// Настраиваемся на работу с BLOB
 tt->SaveToFile("AAA.AAA");
// Записываем данные BLOB в файл
```

Запись из файла в BLOB

Запись данных в BLOB из файла с использованием объекта **TTable** иллюстрируется следующим примером:

Пример 6.18

```
. . .              // Находим нужную строку
TBLOBField * tt=
 dynamic_cast<TBLOBField *>
(Table1->FieldByName("REFERAT"));
// Настраиваемся на работу с BLOB
Table1->Edit();
// Переводим таблицу в режим редактирования
tt->LoadFromFile("AAA.AAA");
// Записываем данные из файла в BLOB
```

При использовании для записи данных в **BLOB** объекта TQuery код примет следующий вид:

```
// Запрос должен быть обновляемым !!
// то есть свойство RequestLive=true

• • .                        // Находим нужную строку
TBLOBField * tt=
 dynamic_cast<TBLOBField *>
(Query1->FieldByName("REFERAT"));
// Настраиваемся на работу с BLOB
Query1->Edit();
// Переводим запрос в режим редактирования
tt->LoadFromFile("AAA.AAA");
// Записываем данные из файла в BLOB
```

Если запрос не является обновляемым, то необходимо выдать явно команду UPDATE. Это можно сделать многими способами. Например, можно поступить так.

Установить свойство TQuery CashedUpdate=true. В этом случае вносимые в поля запроса изменения не посылаются в базу, а просто запоминаются на клиенте. Для внесения изменений необходимо выполнить явным образом команду обновления.

Рассмотрим подобное обновление на следующем примере.

Пример 6.19

Для реализации дополнения необходимо выполнить ряд действий.

- Добавить в форму объект TUpdateSQL, в котором хранятся тексты SQL для внесения изменений. Для них предусмотрены 3 свойства: DeleteSQL, ModifySQL и InsertSQL.
- Связать объект TQuery с TUpdateSQL, задав в свойстве объекта TQuery UpdateObject имя объекта TUpdateSQL. В нашем случае UpdateSQL1.
- В свойства объекта UpdateObject 1 записать запросы на изменение данных:

DeleteSQL

```
delete from TBOOK where  UNIKEY = :OLD_UNIKEY
```

ModifySQL

```
update TBOOK set
  BOOKNM = :BOOKNM,
  REFERAT = :REFERAT
where  UNIKEY = :OLD_UNIKEY
```

InsertSQL

```
insert into TBOOK  (MATHERKEY, BOOKNM, REFERAT)
values   (MATHERKEY, :BOOKNM, :REFERAT)
```

Для нашего примера достаточно только свойства ModifySQL, остальные приведены чисто справочно. Значения параметров запросов на изменение данных автоматически берутся на основе значений текущей строки родительского запроса Query1.

В этом случае процедура обновления данных из файла AAA.AAA будет отличаться от предыдущего примера всего на одну строку и примет вид:

```
TBLOBField * tt=
 dynamic_cast<TBLOBField *>
(Query1->FieldByName("REFERAT"));
// Настраиваемся на работу с BLOB
Query1->Edit();
// Переводим запрос в режим редактирования
```

Расширенные возможности для работы с базой 163

```
tt->LoadFromFile("AAA.AAA");
// Записываем данные из файла в BLOB
//  (пока только на клиенте)
UpdateSQL1->Apply(ukModify);
// Записываем в базу
```

Запись данных из BLOB в поток для последующего чтения

В рассматриваемых выше примерах данные BLOB считывались и обрабатывались либо специальными объектами, либо внешними программами (по результатам записи в файл). Если же данные BLOB необходимо обработать непосредственно, то лучше и чтение и запись производить в соответствующие рабочие переменные.

Пример 6.20

```
. . .                          // Находим нужную строку
TBLOBField * tt=
 dynamic_cast<TBLOBField *>
 (Query1->FieldByName("JOB_REQUIREMENT"));
// Настраиваемся на работу с BLOB
TBLOBStream *bs = new TBLOBStream(tt, bmRead);
// Создаем поток для чтения из выбранного BLOB
Memo1->Lines->LoadFromStream(bs);
// Читаем данные из потока в поле приложения
delete bs; // удаляем поток
```

Чтение данных в BLOB из потока

Пример 6.21

```
. . .
Query1->Edit();
// Переводим запрос в режим редактирования
 TBLOBField * tt=dynamic_cast<TBLOBField *>
 (Query1->FieldByName("JOB_REQUIREMENT"));
// Настраиваемся на работу с BLOB
 TMemoryStream *bs = new TMemoryStream();
 Memo1->Lines->SaveToStream(bs);
// Заполняем поток данными из приложения
 tt->LoadFromStream(bs);
// Записываем данными в BLOB  .
 delete bs; // удаляем поток
. . .
// Выдаем команду на обновление базы (см. пример 6.19)
```

Фактически, поскольку настроечные операции, связанные с открытием таблиц, запросов выполняются однократно, собственно доступ к данным BLOB реализуется одним или двумя операторами. Причем весь кон-

троль правильности выполнения действий по доступу к BLOB выполняется стандартными средствами, что снижает вероятность ошибок.

6.3. Функции пользователя (UDF)

НАЗНАЧЕНИЕ И ВОЗМОЖНОСТИ ИСПОЛЬЗОВАНИЯ

Пользовательские функции (User defined functions - UDF) - это программы на базовом языке для выполнения в приложениях, часто используемых при работе с базой данных задач. UDF помогают обеспечить модульность приложений, выделяя их в отдельные модули многократного использования.

К UDF и фильтрам BLOB можно обращаться через isql или из программ на базовом языке. К UDF можно также обращаться из хранимых процедур и триггеров.

Замечание. UDF и фильтры BLOB не поддерживаются на серверах NetWare.

Пользовательские функции не являются частью базы. Они помещаются в отдельную библиотеку (DLL), находящуюся на той же машине, что и база данных. Чтобы сделать функции доступными в базе, их надо в ней объявить. После этого обращение к ним осуществляется точно так же, как к стандартным функциям InterBase. Другими словами, вызов этих функций может производиться везде, где синтаксисом SQL предусмотрено использование выражений.

Чтобы можно было использовать внешнюю функцию, она должна быть объявлена в базе как внешняя функция, а реализующий ее программный код помещен в библиотеку, указанную в объявлении.

ОБЪЯВЛЕНИЕ ВНЕШНЕЙ ФУНКЦИИ

Как только UDF была написана и откомпилирована в библиотеку, она должна быть объявлена в базе данных, где предполагается ее использовать. Для этого предусмотрена команда DELCARE EXTERNAL FUNCTION.

Каждая функция в базе данных должна быть объявлена отдельно. Пока точка входа и имя модуля (библиотеки) и путь к нему не изменяется, нет необходимости в переобъявлении функции, даже если сама функция изменяется.

Синтаксис объявления внешней функции следующий:

```
DECLARE EXTERNAL FUNCTION name
[LIST_datatypes]
RETURNS {datatype [BY VALUE] / CSTRING ( int)} [FREE_IT]
ENTRY_POINT 'entryname'
```

```
MODULE_NAME 'modulename';

datatypes ;:= {datatype / CSTRING ( int) }
```

datatype - любой разрешенный в InterBase тип данных кроме массивов и BLOB

CSTRING (int) - специальный тип для представления строк, представляющий собой последовательность символов, заканчивающуюся двоичным 0.

FREEJT - специальная конструкция, обеспечивающая указывающая на необходимость освобождения памяти выделенной пользовательской функцией для возвращаемого значения, передаваемого по ссылке. Необходимость в такой конструкции вызвана тем, что запрос памяти производится в **UDF** , а освобождение должно выполнятся в InterBase, о чем, конечно, должно быть известно заранее.

Таблица 6.6. *Синтаксические конструкции объявления внешних функций*

Конструкция	*Описание*
name	Имя UDF для использования в командах SQL; может отличаться от имени функции (в библиотеке) указанной после ключевого слова ENTRY_POINT
datatype	Тип данных входного параметра или возвращаемого значения. Все входные параметры передаются UDF по ссылке. Возвращаемое значения может передаваться как по ссылке, так и по значению. Использование элементов массива запрещено
RETURNS	Определяет возвращаемое функцией значение
BY VALUE	Указывает, что возврат производится по значению. Если конструкция опущена, то возврат производится по ссылке
CSTRING (int)	Указывает, что UDF возвращает строку, заканчивающуюся ограничителем \0
FREEJT	Освобождает память, занятую возвращаемым значением после выполнения UDF. Использовать только, если память в UDF выделялась динамически
'entryname'	Заключенное в кавычки имя UDF, как оно записано в библиотеке
'modulename'	Спецификация файла, идентифицирующая библиотеку, которая содержит UDF. Текст должен заключаться в кавычках.

	* Библиотека должна постоянно находиться на сервере; путь должен относиться к местоположению библиотеки на сервере. * На любой платформе модуль может быть указан без имени пути, если он находится в ib_install_dir/lib. * Следует использовать полное имя файла библиотеки, включая расширение, даже если имя пути не указано

СОЗДАНИЕ ПОЛЬЗОВАТЕЛЬСКИХ ФУНКЦИЙ

Тот факт, что в InterBase включен только базовый набор функций для использования в SQL-выражениях, ни в коей мере не является ограничением. Механизм User Defined Functions (UDF), позволяет писать пользовательские функции на любом компилирующем инструменте разработки (базовом языке). Такие функции выполняются на сервере, причем в рамках процесса сервера, что повышает скорость их вызова практически до уровня скорости вызова стандартных SQL-функций.

UDF могут быть написаны, вообще говоря, на любом языке. Здесь мы рассмотрим подготовку функций на C (C++). Прежде всего, отметим, что все передаваемые в UDF параметры всегда передаются по ссылке, то есть должны быть объявлены в функции как указатели. Возвращаемое значение может передаваться и по ссылке и по значению.

Рассмотрим процесс создания библиотеки функций пользователя. Детали процесса зависят, вообще говоря, от используемого компилятора. Построим такую библиотеку для Windows в среде C++Builder. Воспользуемся DLL Wizard'ом, необходимым для корректной настройки компоновщика в файле проекта. В других компиляторах проект будет строиться иначе, но сам текст программ будет идентичен. В начале текста подключается заголовочный файл <windows.h>, за которым объявляются наши функции (их объявления можно также поместить в свой заголовочный файл).

Пример 6.22

```
//---------------------------------------------------------------
#include <windows.h>
extern "C" long __declspec(dllexport)
  if_i(double* m,long* a,long* b);
extern "C" double *__declspec(dllexport)
  if_d(double* m,double* a,double* b);
extern "C" char *__declspec(dllexport)
  r_upper(char*a);
extern "C" char *__declspec(dllexport)
  rupper(char*a);
extern "C" int __declspec(dllexport)
  idate(int*a);
```

```
#pragma argsused
int WINAPI DllEntryPoint(HINSTANCE hinst,
  unsigned long reason, void* lpReserved)
{
  return 1;
}
//---------------------------------------------------------------
. . . . . .
```

В объявлении функций стоит обратить внимание на конструкции *extern "C"* и *__declspec(dllexport)*. Эти конструкции обеспечивают корректность хранения имен и вызова функций из создаваемой DLL. Далее идет текст самих функций. Остановимся на нем несколько подробнее.

Первая функция *if_i* является аналогом условной операции С. Она возвращает второй аргумент, если значение первого положительно, и третий - в противном случае.

```
long__declspec(dllexport)
    if_i(double* m,long* a,long* b)
{return (*m>0)?*a:*b;}
```

Соответствующее ей объявление внешней функции в базе будет иметь такой вид:

```
DECLARE EXTERNAL FUNCTION IIF
DOUBLE PRECISION, INTEGER, INTEGER
RETURNS INTEGER BY VALUE
 ENTRY_POINT "if_i"
 MODULE_NAME "e:\USERSYS\MDLL\MyDLL";
```

Здесь "e:\USERSYS\MDLL\MyDLL" задает полный путь к DLL.

Если библиотека помещена в стандартную директорию InterBase, то путь к ней указывать не надо.

```
DECLARE EXTERNAL FUNCTION IIF
DOUBLE PRECISION, INTEGER, INTEGER
RETURNS INTEGER BY VALUE
 ENTRY_POINT "if_i"
 MODULE_NAME "MyDLL";
```

Имена функции в базе и в библиотеке, как видно из примера, могут и не совпадать. Кроме того, нужно помнить, что имена функций в библиотеке зависят от регистра, а в базе - нет.

Замечание

В последних версиях InterBase местоположение DLL предопределено, это одна из поддиректорий директории, где размещен InterBase после его установки. Обычно при инсталляции устанавливается поставляемая с InterBase библиотека UDF - udf_lib.dll. Она всегда помещена «там, где надо», так что все пользовательские библиотеки надо размещать там же.

Функции типа IIF весьма полезны в тех случаях, когда выбираемое из базы значение зависит от некоторого условия. В определенном смысле даже странно, что они не входят в состав стандартной библиотеки UDF.

Например, в таблице T поле A может быть положительным или отрицательным.

Пример 6.23

Таблица 6.7. *Содержимое полей A в таблице T*

A
7
-6
12
-4
5
-3

В итоговой таблице нужно просуммировать отдельно положительные и отрицательные значения (типичным примером может служить суммирование прихода и расхода на складах). Такой запрос с помощью нашей функции можно записать непосредственно:

```
SELECT  sum(IIF(A, A, 0)) PLUS,
        Sum(IIF(A, 0, -A)) MINUS
FROM  T
```

Таблица 6.8. *Результат выборки с помощью функции IIF*

PLUS	*MINUS*
24	13

Функция *if_i* возвращает результат по значению. В этом случае важно точное совпадение типов данных, объявленных для функции в базе и фактически используемых функцией.

Рассмотрим теперь следующую функцию, выполняющую те же действия, но для чисел с плавающей точкой.

Пример 6.24

```
double *__declspec(dllexport)
    if_d(double* m,double* a,double* b)
{ return (*m>0)?a:b;}
```

Здесь возвращается не значение, а указатель на него, то есть мы имеем случай возврата по ссылке.

Соответствующее объявление внешней функции в базе будет иметь такой вид:

```
DECLARE EXTERNAL FUNCTION DIF
DOUBLE PRECISION, DOUBLE PRECISION, DOUBLE PRECISION
RETURNS DOUBLE PRECISION
 ENTRY_POINT "if_d"
 MODULE_NAME "MyDLL";
```

Следует обратить внимание на то, что при обращении к DLL ей передаются указатели и результатом также является указатель. Соответствие типов указателей не проверяется. Последнее означает, что данная функция может быть использована и для получения результатов другого типа.

Рассмотрим объявление внешней функции CIF.

Пример 6.25

```
DECLARE EXTERNAL FUNCTION CIF
DOUBLE PRECISION, VARCHAR(256), VARCHAR(256)
RETURNS VARCHAR(256)
 ENTRY_POINT "if_d"
 MODULE_NAME "MyDLL";
```

И сразу же рассмотрим SQL-запрос

```
SELECT  DIF(A, A, 0) PLUS, DIF(A, 0, -A) MINUS,
 CIF(A, 'Положител.','Отрицат.') Text
FROM  T
```

В конечном счете, получим:

Таблица **6.9.** *Результат выборки с помощью функций* ***DIF, CIF***

PLUS	*MINUS*	*TEXT*
7,000	0,000	Положител.
0,000	6,000	Отрицат.
12,000	0,000	Положител.
0,000	4,000	Отрицат.
5,000	0,000	Положител.
0,000	3,000	Отрицат.

Здесь результатом должна являться символьная строка, хотя функция работает вроде бы с числами с плавающей точкой. На самом деле, плавающая точка тут не причем. Используются только указатели. Так что наша функция ***if_d*** не зависит от типов данных, указатель на которые она возвращает. В этом смысле честнее было бы написать ее в виде

```
void *__declspec(dllexport)
    if_d(double* m, void * a, void * b)
{return (*m>0)?a:b;}
```

Следующие функции иллюстрируют работу со строковыми данными. Функции ***r_upper*** и ***rupper*** обеспечивают перевод алфавитных данных в кодировке WIN 1251 из нижнего регистра в верхний как для латыни, так и для кириллицы. Эта функция может быть полезна в том случае, если кодовая страница для базы не указана (NONE). Пара функций приведена для того, чтобы проиллюстрировать различные методы обработки параметров и возвращаемых величин.

Пример 6.26

```
char *__declspec(dllexport) r_upper(char *a){
 char *c,t;
 int i,n=(short)*a;
 c=(char *)malloc(256);
 for(i=0;i<n;i++){
  t=a[i+2];
  if(t>='a' && t<='z') t=t+'A'-'a';
  else if(t>='а' && t<='я') t=t+'A'-'я';
  c[i]=t;
 }
 c[n]=0;
 return c;
}
```

Для формирования выходного параметра необходимо выщеление области памяти. Статическое выделение при многопрограммной работе не подходит, поскольку в этом случае нельзя гарантировать, что память используемая одним запросом не будет испорчена другим. Автоматическая память (стековая) не годится для передачи по ссылке, поскольку она будет освобождена сразу же по выходе из программы. Остается только явное выделение памяти в программе. Для этого используется функция *malloc*. Последнее существенно потому, что выделенная память должна освобождаться уже в InterBase. Следовательно, механизмы выделения и освобождения памяти должны быть согласованы, а механизм работы *malloc* как раз такой, какой используется в InterBase.

И еще одно замечание. Объем выделяемой в программе памяти должен согласовываться с размером возвращаемого аргумента в объявлении внешней функции в базе.

В ряде случаев можно обойтись без явного выделения памяти. Для этого следует использовать память, занимаемую входными параметрами.

Пример 6.27

```
char *___declspec(dllexport) srupper(char *a){
 unsigned char t;
 int i;
 for(int i=0;a[i];i++){
  t=a[i];
  if(t>='a' && t<='z') t=t+'A'-'a';
  else if(t>=224) t-=32;
  a[i]=t;
 }
 return a;
}
```

Особенности представления данных в InterBase

При работе с данными различных типов необходимо учитывать особенности их представления в InterBase.

Таблица 6.10. *Соответствие типов данных в InterBase и C*

Тип InterBase	*Объявление данных в C (C++)*
SMALLINT	short int p;
INTEGER	int p;
FLOAT	float p;
DOUBLE PRECISION	double p;

Тип InterBase	*Объявление данных в C (C++)*
NUMERIC(m, n) / DECIMAL(m, n)	если m<5, то **short int** p; если 5≤m<10, то **int** p; Для версий до 6 если m≥10, то **double** p Для версий от 6.0 если 10≤m≤18, то **int_64 p;** значение задается с масштабирующим множителем 10^{-n}
DATE	массив из 2 **int** p[2]; p[0] - количество дней с 17 ноября 1858; p[1] - количество секунд* 10000 от начала суток
CHARACTER(n) / CHAR(n)	**char** p[n]; (двоичный 0 в конце не предполагается, заполняются пробелами справа до явно указанной длины)
VARYING CHARACTER / VARCHAR(n)	**struct /** **short int** p1; **char** p2[n]; **} p;** p.p1 - фактическая длина поля; p.p2 - символьная строка (двоичный 0 в конце не предполагается)
CSTRING(n)	**char** p[n+1]; (с двоичным 0 **в** конце в качестве ограничителя)
BLOB	см. ниже пример UDF для работы с BLOB

ПримерUDFдляработысBLOB

Данная функция читает содержимое BLOB и записывает его в файл COPYBLOB.ooo. В данном случае используется явный вызов функций API, о котором говорилось в разделе 6.2, для доступа к данным InterBase. Естественно, что аналогичные средства можно применять не только в UDF, но и в любой прикладной программе, хотя в прикладных программах, по-моему, проще использовать тот сервис, который представляется средой разработки.

Пример 6.26

```
#include <stdio.h>
// описание управляющей структуры BLOB
```

```
struct blobs (
 void  (*blob_get_segment)(int *,char *,long,  long&);
 int *blob_handle;
 long number_segments;
 long max_seglen;
 long total_size;
 void  (*blob_put_segment)();


typedef struct blobs* SBLOB;

extern "C" int___declspec(dllexport)
  pr_blob(SBLOB bl) ;

int___declspec(dllexport) pr_blob(SBLOB bl)
{
 long length=100;
 char *buffer;
 FILE *out;
 if(!(bl->blob_get_segment)) return 0;
// проверка, что действительно передан BLOB
 if((out=fopen("COPYBLOB.ooo", "wb"))==NULL)
   return -1;
// проверка успешности создания файла
 buffer=new char[bl->max_seglen+1] ;
// создание буфера для чтения
 for(int i=0;i<bl->number_segments;i++)
 {
// цикл чтения BLOB и записи его в файл
 (bl->blob_get_segment)(bl->blob_handle,buffer,
    bl->max_seglen,length);
  fwrite(buffer, 1, length, out);
 }
 fclose(out);
 delete[] buffer;
 return 1;
}
```


```
// цикл чтения BLOB и записи его в файл
  (bl->blob_get_segment)(bl->blob_handle,buffer,
    bl->max_seglen,length);
  fwrite(buffer, 1, length, out);
 fclose(out);
 delete[] buffer;
 return 1;
}
```


6.4. Фильтры BLOB

Фильтры BLOB (Binary Large Object - большой двоичный объект) - это программы на базовом языке, которые конвертируют данные BLOB из одного формата в другой. К фильтрам BLOB можно обращаться через isql или прикладные программы на базовом языке.

Отметим, что [illegible] иллюстрирует механизм чтения данных из BLOB.

Замечание. Фильтры BLOB не поддерживаются на серверах NetWare.

ФИЛЬТРАЦИЯ ДАННЫХ BLOB

Понимание роли подтипов BLOB особенно важно при работе с фильтрами BLOB. Фильтр BLOB - подпрограмма, которая транслирует данные BLOB одного подтипа в другой. InterBase включает набор специальных внутренних фильтров BLOB, которые конвертируют подтип О в подтип 1 (текст), и подтип 1 (текст) в подтип 0. В дополнение к стандартным фильтрам можно запрограммировать собственные внешние фильтры для обеспечения конвертирования других типов данных. Например, можно запрограммировать фильтр для трансляции растровых изображений или медиаданных одного формата в другой (другой вопрос, для чего это нужно делать именно в базе).

Замечание

Фильтры BLOB можно использовать для баз данных на всех платформах сервера InterBase, кроме Netware, где фильтры BLOB не поддерживаются.

Использование стандартных текстовых фильтров InterBase

Стандартные фильтры InterBase конвертируют данные BLOB подтипа 0 или любого другого системного типа InterBase в подтип 1 (текст).

Когда текстовый фильтр используется для чтения данных столбца BLOB, он меняет стандартное поведение InterBase для обработки сегментов. Независимо от фактического характера сегментов в столбце BLOB, текстовый фильтр устанавливает правило, что сегменты должны закончиться символом перевода строки (\n).

Текстовый фильтр возвращает все символы, включая первый перевод строки как первый сегмент, следующие символы, включая второй перевод строки, как второй сегмент и так далее.

Для конвертирования любого нетекстового подтипа в текст следует объявлять его FROM-подтип как подтип 0, а его TO-подтип как подтип 1.

Использование внешних фильтров BLOB

В отличие от стандартных фильтров InterBase, которые выполняют трансляцию подтипов 0 и 1, внешний фильтр BLOB - вообще часть библиотеки подпрограмм, которая создается для конкретных приложений.

Чтобы использовать внешний фильтр, необходимо сначала запрограммировать его, откомпилировать, поместить в библиотеку (DLL в системе Windows), затем объявить в базе данных, содержащей данные BLOB, которые должны обрабатываться.

Объявление внешних фильтров в базе данных

Для объявления внешних фильтров в базе данных используется команда DECLARE FILTER. Например, следующая команда объявляет фильтр TFILTER:

Пример 6.27

```
EXEC SQL
DECLARE FILTER TFILTER
INPUT_TYPE -1 OUTPUT_TYPE -2
ENTRY_POINT "_TFilter"
MODULE_NAME "MYDLL.dll";
```

В примере, входной подтип фильтра определен как -1, выходной как -2. В этом примере, INPUTTYPE определяет текст нижнего регистра, а OUTPUT_TYPE определяет текст верхнего регистра. Цель фильтра TFILTER, таким образом, состоит в том, чтобы конвертировать данные BLOB из текста в нижнем регистре в текст верхнего регистра.

Параметры ENTRY_POINT и MODULE_NAME определяют внешнюю подпрограмму, которую InterBase вызывает, когда запрашивается фильтр. Параметр MODULE_NAME определяет MYDLL.dll, динамически загружаемую библиотеку, содержащую выполнимый код фильтра. Параметр ENTRY_POINT определяет точку входа в DLL. В примере указана та же библиотека, что и для UDF, но можно поместить фильтры и в другую. Никаких ограничений на этот счет нет.

Использование фильтров для чтения и записи данных

Следующая схема показывает заданное по умолчанию поведение фильтра TFILTER, который преобразует текст из нижнего регистра в верхний.

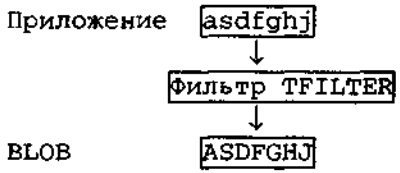


Рис. 6.3. Схема работы фильтра BLOB (строчные в прописные).

Точно так же при чтении данных фильтр TFILTER может читать данные BLOB подтипа -2 и преобразовать их в подтип -1.

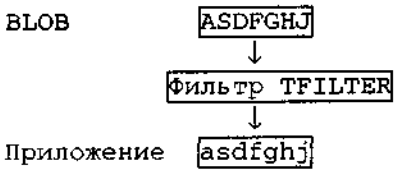


Рис. 6.4. Схема работы фильтра BLOB (прописные в строчные).

Вызов фильтра в приложении

Для вызова фильтра в приложении используется опция FILTER при объявлении курсора BLOB. Тогда при выполнении приложением операций, использующих курсор, InterBase автоматически вызывает фильтр.

Например, следующее определение курсора INSERT означает, что фильтр SAMPLE, должен использоваться в любых операциях с курсором BCINS1.

Пример 6.28

```
EXEC SQL
DECLARE BCINS1 CURSOR FOR
INSERT BLOB BLOB1 INTO TABLE1
FILTER FROM -1 TO -2;
```

Когда InterBase обрабатывает это объявление, он ищет в списке фильтров, определенных в текущей базе данных, фильтр с соответствием подтипов FROM и TO. Если такой фильтр существует, InterBase вызывает его при операциях с BLOB, использующих курсор BCINS1. Если InterBase не может найти такой фильтр, приложению возвращается сообщение об ошибке.

НАПИСАНИЕ ВНЕШНИХ ФИЛЬТРОВ

Для написания собственных фильтров необходимо детальное понимание структуры транслируемых данных. InterBase не выполняет проверки данных BLOB, но требует совместимости типов исходного и результирующего BLOB. Поддержание совместимости лежит на разработчике фильтра.

Типы фильтров

Фильтры разделяются на два типа: те, которые преобразовывают данные по одному сегменту, и те, которые преобразовывают данные множества сегментов одновременно.

Фильтры первого типа читают сегмент данных, преобразовывают его и возвращают результат приложению, второго - читают все данные при открытии курсора BLOB, выполняют все преобразование, а затем имитируют посегментную передачу их приложению. Если время и этапность обработки существенны для приложения, следует тщательно рассмотреть возможности применения того из типов, который в большей степени соответствует поставленной цели.

Read-only и write-only фильтры

Некоторые фильтры могут поддерживать только чтение из BLOB или только запись в BLOB, но не обе операции. При попытке использовать фильтр BLOB для операции, которую он не поддерживает, InterBase возвращает приложению сообщение об ошибке.

Создание функций фильтров

При создании фильтра необходимо указать точку входа, задающую имя функции фильтра в разделе объявлений программы. InterBase вызывает функцию фильтра, когда приложение выполняет операции с BLOB. Все связи между InterBase и фильтром реализуются функцией фильтра. Функция самого фильтра может вызывать другие функции, которые необходимы для выполнения программы фильтра. Далее фильтр должен быть объявлен в базе данных командой DECLARE FILTER с указанием параметров ENTRY_POINT в MODULE_NAME.

Функция фильтра должна иметь следующую сигнатуру:

```
filter_function_name(short action, isc_blob_ctl control);
```

Параметр *action* определяет одно из восьми возможных макроопределений действия, параметр *control* определяет управляющую структуру isc_blob_ctl данных BLOB. Определение структуры *isc_blob_ctl* дано в заголовочном файле InterBase ibase.h. Эти параметры будут рассмотрены ниже. Приведем теперь основные декларативные элементы функции фильтра (*jpeg_filter*).

Пример 6.39

```
#include <ibase.h>
#define SUCCESS 0
#define FAILURE 1
ISC_STATUS jpeg_filter(short action, isc_blob_ctl control)
{
 ISC_STATUS status = SUCCESS;
 switch (action)
 {
  case isc_blob_filter_open:
```

```
    . . .
    break;
  case isc_blob_filter_get_segment:
    . . .
    break;
  case isc_blob_filter_create:
    . . .
    break;
  case isc_blob_filter_put_segment:
    . . .
    break;
  case isc_blob_filter_close:
    . . .
    break;
  case isc_blob_filter_alloc:
    . . .
    break;
  case isc_blob_filter_free:
    . . .
    break;
  case isc_blob_filter_seek:
    . . .
    break;
  default:
    status = isc_uns_ext /* unsupported action value */
    . . .
    break;
 }
 return status;
}
```

InterBase передает одно из восьми возможных действий (параметр *action)* функции фильтра ***jpeg_filter()*** и экземпляр управляющей структуры BLOB (параметр *isc_blob_ctl).*

Многоточия (...) в приведенном листинге заменяют текст программы, реализующий действительную обработку данных для каждого из возможных действий фильтра. С каждым действием связан блок case, соответствующий операции с базой данных, которая может потребоваться приложению.

lsc_blob_ctl - управляющая структура BLOB; содержит основные данные для управления обменом между InterBase и фильтром.

Определение управляющей структуры BLOB

Управляющая структура *isc_blob_ctl* обеспечивает базовые методы обмена данными между InterBase и фильтром. Объявление управляющей структуры находится в заголовочном файле InterBase ibase.h. Для его подключения к программе используется инструкция

```
#include <ibase.h>.

typedef struct isc_blob_ctl{
 ISC_STATUS       (ISC_FAR *ctl_source)();
/* Source filter */
 struct isc_blob_ctl ISC_FAR *ctl_source_handle;
/* Argument to pass to source */
/* filter */
 short     ctl_to_sub_type;     /* Target type */
 short     ctl_from_sub_type;   /* Source type */
 unsigned short  ctl_buffer_length;  /* Length of buffer */
 unsigned short  ctl_segment_length; /* Length of current
segment */
 unsigned short    ctl_bpb_length;   /* Length of blob pa-
rameter */
/* block */
 char   ISC_FAR *ctl_bpb; /* Address of blob parameter */
/* block */
 unsigned char ISC_FAR *ctl_buffer; /* Address of segment
buffer */
 ISC_LONG       ctl_max_segment;/* Length of longest seg-
ment */
 ISC_LONG  ctl_number_segments; /* Total number of segments
*/
 ISC_LONG    ctl_total_length; /* Total length of blob */
 ISC_STATUS  ISC_FAR *ctl_status;/* Address of status vec-
tor */
 long  ctl_data [8];         /* Application specific data
*/
} ISC_FAR *ISC_BLOB_CTL;
```

Структура *isc_blob_ctl* используется двумя способами:

1. Когда приложение исполняет операцию доступа к BLOB, InterBase вызывает функцию фильтра и передает ей экземпляр *isc_blob_ctl.*
2. Функции фильтра могут передавать экземпляр *isc_blob_ctl* подпрограммам доступа к данным InterBase.

В любом случае назначение некоторых полей *isc_blob_ctl* зависит от выполняемого действия (параметр action). Например, когда приложение делает попытку вставки BLOB, InterBase передает функции фильтра действие-вставку (параметр *action = isc_blob_filter_put_segment*). Функция фильтра передает экземпляр управляющей структуры InterBase.

Ctl_buffer структуры содержит сегмент данных, которые должны быть записаны согласно команде INSERT BLOB приложения. Поскольку буфер содержит информацию для передачи в функцию, это поле является входным. В случае *isc_blob_filter_put_segment,* то есть для выполнения записи в базу данных, функция фильтра должна включить команды в конструкции case.

В другом случае, например, когда приложение делает попытку выполнить команду FETCH, действие - выборка (параметр *action* = *isc_blob_filter_get_segment*). В функцию фильтра должна включаться группа команд для заполнения *ctl_buffer* данными сегмента из базы данных для его возврата приложению, вызвавшему функцию. Здесь буфер используется в функции фильтра как выходной.

В таблице ниже описываются поля управляющей структуры BLOB isc_blob_ctl и характер их использования в функции фильтра (входные - IN, выходные - OUT).

Таблица 6.10. *Управляющая структура BLOB isc_blob_ctl*

Имя поля	*Описание*
(*ctl_source)()	Указатель на внутреннюю подпрограмму InterBase доступа к BLOB. (IN)
*ctl_source_handle	Указатель на экземпляр isc_blob_ctl, который передается внутренней подпрограмме InterBase доступа к BLOB. (IN)
ctl_to_sub_type	Указывает подтип BLOB-результата. Информационное поле. Необходимо для многоцелевых фильтров, которые могут исполнять несколько видов преобразований. Это и следующее поля дают возможность такому фильтру определить, какую именно трансляцию следует выполнить. (IN)
ctl_frora_sub_type	Указывает подтип BLOB-источника. Информационное поле. Необходимо для многоцелевых фильтров, которые могут исполнять несколько видов преобразований. Это и предыдущее поля дают возможность такому фильтру определить, какую именно трансляцию следует выполнить. (IN)
ctl_buffer_length	Для isc_blob_filter_put_segment поле - входное (IN), содержащее длину сегмента данных в ctl_buffer. Для isc_blob_filter_get_segment поле - входное (IN), устанавливающее размер буфера (адресованного ctlbuffer) для сохранения полученных из BLOB данных
ctl_segment_length	Длина текущего сегмента. Это поле не используется для isc_blob_filter_put_segment. Для isc_blob_filter_get_segment поле - выходное (OUT) устанавливается в длину сегмента полученных из BLOB данных (или части сегмента, если длина буфера ctl_buffer_length меньше фактической длины сегмента)
ctl_bpb_length	Длина буфера параметров BLOB. Зарезервировано для будущего расширения

Имя поля	*Описание*
*ctl_bpb	Указатель на буфер параметров BLOB. Зарезервировано для будущего расширения
*ctl_buffer	Указатель на буфер сегмента. Для isc_blob_filter_put_segment поле - входное (IN). Содержит данные сегмента. Для isc_blob_filter_get_segment поле - выходное (OUT), функция фильтра заполняет его данными сегмента для возврата вызвавшему фильтр приложению
ctl_max_segment	Длина самого большого сегмента в BLOB. Начальное значение - 0. Это поле устанавливает функция фильтра. Поле только информационное
ctl_number_segments	Начальное значение - 0. Это поле устанавливает функция фильтра. Поле только информационное
ctl_total_length	Полная длина BLOB. Начальное значение - 0. Это поле устанавливает функция фильтра. Поле только информационное
*ctl_status	Указатель на вектор состояния InterBase. (OUT)
ctl_data[8]	Массив из 8 элементов. Зависит от приложения. Можно использовать это поле, например, для хранения указателей на ресурсы, типа указателей памяти и дескрипторов файла, созданных обработчиком isc_blob_filter_open. Тогда при следующем вызове функции фильтра указатели ресурсов будут доступны для использования. (IN/OUT)

Установка значений полей управляющей структуры

Структура isc_blob_ctl содержит три поля, сохраняющие информацию о BLOB, к которому осуществляется доступ: ctl_max_segment, ctl_number_segments и ctl_total_length.

Необходимо контролировать правильность значений этих полей в функции фильтра всегда, когда это возможно. В зависимости от назначения фильтра поддержка правильности значений этих полей не всегда возможна. Например, фильтр, который сжимает данные посегментно, не может определять размер ctl_max_segment, пока не обработаны все сегменты.

Эти поля носят только информационный характер. InterBase не использует значения этих полей во внутренней обработке.

ПРОГРАММИРОВАНИЕДЕЙСТВИЙФУНКЦИИФИЛЬТРА

Когда приложение выполняет операцию доступа к BLOB, InterBase передает функции фильтра соответствующее сообщение о действии в па-

раметре *action*. Имеются восемь возможных действий, каждое из которых является следствием специфической операции доступа. Следующий список макроопределений действий объявлен в заголовочном файле ibase.h:

```
#define isc_blob_filter_open        0
#define isc_blob_filter_get_segment 1
#define isc_blob_filter_close       2
#define isc_blob_filter_create      3
#define isc_blob_filter_put_segment 4
#define isc_blob_filter_alloc       5
#define isc_blob_filter_free        6
#define isc_blob_filter_seek        7
```

Приводимая таблица описывает операции доступа к BLOB, которые соответствуют каждому действию (параметр action).

Таблица 6.10. *Операции доступа к BLOB в зависимости от параметра action*

Действие	*Условие вызова*	*Назначение*
isc_blob_filter_open	Приложение открывает BLOB курсор на чтение	Установка информационных полей управляющей структуры BLOB. Выполняет задачи инициализации, типа распределения памяти или открытия временных файлов. Устанавливает в случае необходимости переменную состояния. Значение переменной состояния становится возвращаемым значением функции фильтра
isc_blob_filter_get_segment	Приложение выполняет команду FETCH для BLOB	Заполнение полей ctl_buffer и ctl_segment_length управляющей структуры BLOB содержанием сегментов отранслированных данных для возврата функцией фильтра. Выполняет конвертирование данных, если фильтр обрабатывает BLOB посегментно. Устанавливает переменную состояния. Ее значение становится возвращаемым значением функции фильтра

Действие	*Условие вызова*	*Назначение*
isc_blob_filter_close	Приложение закрывает курсор BLOB	Выполняется задача выхода, типа, освобождения распределенной памяти, закрытия или удаления временных файлов
isc_blob_filter_create	Приложение открывает курсор вставки BLOB	Установка информационных полей управляющей структуры BLOB. Выполняются задачи инициализации, типа, распределения памяти или открытия временных файлов. Устанавливается в случае необходимости переменная состояния. Значение переменной состояния становится возвращаемым значением функции фильтра
isc_blob_filter_put_segment	Приложение выполняет команду INSERT для BLOB	Выполняется конвертирование данных сегмента, передаваемых через управляющую структуру BLOB. Запись данных сегмента в базу данных. Если процесс трансляции изменяет длину сегмента, новое значение длины должно быть отражено в параметрах, передаваемых функции записи. Устанавливает переменную состояния. Значение переменной состояния становится возвращаемым значением функции фильтра
isc_blob_filter_alloc	InterBase инициализирует работу фильтра; не является результатом действия приложения	Установка информационных полей управляющей структуры BLOB. Выполняются задачи инициализации, типа, распределения памяти или открытия временных файлов. Установка в случае необходимости переменной состояния. Значение переменной состояния становится возвращаемым значением функции фильтра

Действие	*Условие вызова*	*Назначение*
isc_blob_filter_free	InterBase завершает обработку фильтра; не является результатом действия приложения	Выполнение задачи выхода, типа освобождения распределенной памяти, закрытия или удаления временных файлов
isc_blob_filter_seek	Зарезервировано для внутреннего использования фильтра; не используется внешними фильтрами	

Следует сохранять указатели ресурсов, типа указателей памяти и дескриптора файла, созданных обработчиком *isc_blob_filter_open*, в поле *ctl_data* управляющей структуры *isc_blob_ctl* BLOB. Тогда при следующем вызове функции фильтра указатели ресурсов останутся доступными.

Контроль возвращаемых функцией значений

Функция фильтра должна возвращать целое число, указывающее состояние операции, которую она выполнила. Можно построить функцию, возвращающую значения состояния InterBase, даваемые внутренней подпрограммой InterBase.

В некоторых приложениях фильтра функция фильтра должна формировать значения состояния непосредственно. В следующей таблице перечисляются значения состояния, применяемые при обработке BLOB.

Таблица 6.12. *Коды состояния, возвращаемые функциями фильтра*

Константа Макроса	*Величина*	*Содержание*
SUCCESS	0	Указывает, что фильтр отработал успешно. При операции чтения BLOB (isc_blob_filter_get_segment) указывает, что сегмент прочитан полностью
FAILURE	1	Указывает на неудачную операцию. В большинстве случаев состояние более определенно указывает на тип ошибки
isc_uns_ext	См. ibase.h	Указывает, что предпринятое действие не поддерживается фильтром. Например, фильтр только для чтения возвратил бы isc_uns_ext для действия isc_blob_filter_put_segment

Константа Макроса	***Величина***	***Содержание***
isc_segment	См. ibase.h	Указывает, что при операции чтения BLOB выделенный буфер слишком мал для хранения оставшихся байтов текущего сегмента. В этом случае только ctl_buffer_length байтов скопировано, а остаток сегмента должен быть получен через дополнительные запросы isc_blob_filter_get_segment
isc_segstr_eof	См. ibase.h	Указывает, что при операции чтения BLOB был достигнут конец BLOB и нет более никаких дополнительных сегментов для чтения

Глава 7

Организация хранения метаданных

7.1. Назначение и порядок использования описаний данных

В InterBase описания данных, или метаданные, хранятся вместе с пользовательскими данными. Чтобы система заранее могла знать, что это за данные и как ими пользоваться, имена соответствующих таблиц заранее определены. Внешне все объекты системного характера можно легко отличить от пользовательских - они имеют стандартный префикс RDB$; следовательно, имена, создаваемые пользователем, не должны иметь такого префикса. Правда, если об этом не предупреждать заранее, трудно вообразить, что кто-либо стал бы придумывать подобные имена.

Кроме метаданных в InterBase предусмотрено и хранение разного рода комментариев к любым создаваемым пользователем информационным объектам, что обеспечивает возможность хранения документации о базе в самой базе, а это - очень большое удобство. Если Вы хотите создать максимум неудобств для себя и, особенно для тех, кто сопровождает базу данных, никогда не пишите соответствующих комментариев.

Кроме системных таблиц в базе можно создавать и системные обзоры. Автоматически они не создаются, но можно использовать готовый SQL для создания стандартных обзоров, регламентированных стандартом SQL-92. При желании можно также создать и свои собственные обзоры для обеспечения более удобного доступа к описаниям данных и комментариям к ним.

7.2. Системные таблицы

Системные таблицы InterBase содержат метаданные базы данных. Они создаются автоматически сервером InterBase при создании базы данных и изменяются всякий раз, когда выполняются команды, изменяющие структуру данных. Попытка вручную изменять эти данные в случае ее удачи может иметь самые пагубные последствия. Для изменения структуры данных есть специальные команды, их и следует использовать. В то же время информация, содержащаяся в этих таблицах и описывающая таблицы, их поля, домены, триггеры и многое другое, доступна для прикладного программиста на основе обычных SQL запросов и весьма полезна.

Прежде всего, дадим перечень этих таблиц с их описанием.

RDB$CHARACTER_SETS
RDB$CHECK_CONSTRAINTS
RDB$COLLATIONS
RDB$DATABASE
RDB$DEPENDENCIES
RDB$EXCEPTIONS
RDB$FIELD_DIMENSIONS
RDB$FIELDS
RDB$FILES
RDB$FILTERS
RDB$FORMATS
RDB$FUNCTION_ARGUMENTS
RDB$FUNCTIONS
RDB$GENERATORS
RDB$INDEX_SEGMENTS
RDB$INDICES
RDB$LOG_FILES
RDB$PAGES
RDB$PROCEDURE_PARAMETERS
RDB$PROCEDURES
RDB$REF_CONSTRAINTS
RDB$RELATION_CONSTRAINTS
RDB$RELATION_FIELDS
RDB$RELATIONS
RDB$ROLES
RDB$SECURITY_CLASSES
RDB$TRANSACTIONS
RDB$TRIGGER_MESSAGES
RDB$TRIGGERS
RDB$TYPES
RDB$USER_PRIVILEGES
RDB$VIEW_RELATIONS

RDB$CHARACTER_SETS

Описывает доступные для InterBase наборы символов. Общее количество таких наборов - несколько десятков. В таблице задаются комбинации значений по умолчанию. Для использования с русскоязычными данными в среде Widows следует либо указывать NONE, либо применять набор (Character set) Win1251. В первом случае никаких преобразований данных не производится, а сортировки выполняются в порядке возрастания кодов, что обеспечивает алфавитную сортировку текстов сначала по прописным, а затем по строчным буквам. Во втором случае, если необходимо сортировать текстовые данные вне зависимости от регистра, то следует задавать упорядочением (Collation) PXW_CYRL. Изменить непосредственно в таблице значение Rdb$default_collate_name с Win1251 на PXW_CYRL нельзя. Необходимо задавать упорядочение при описании таблиц или доменов.

Таблица 7.1. *Структура набора символов InterBase*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$character_set_name	Char(31)	Наименование кодовой таблицы распознаваемой InterBase
Rdb$form_of_use	Char(31)	Зарезервировано для внутреннего использования
Rdb$number_of_characters	Integer	Число символов кодовой таблицы (например, для японского языка)
Rdb$default_collate_name	Char(31)	Последовательность сравнения для кодовой таблицы по умолчанию
Rdb$character_set_id	Smallint	Уникальный идентификатор кодовой таблицы
Rdb$system_flag	Smallint	Признак типа кодовой таблицы: Определенный пользователем (0 или NULL) Системный (1)
Rdb$description	BLOB 80	Содержит пользовательский комментарий
Rdb$function_name	Char(31)	Зарезервировано для внутреннего использования
Rdb$bytes_per_character	Smallint	Размер символа в байтах

RDB$CHECK_CONSTRAINTS

Содержит данные об ограничениях логической целостности и NOT NULL. Данная таблица может быть полезна для получения информации об используемых в базе ограничениях NOT NULL (кроме явно описанных при задании доменов).

Таблица 7.2. *Ограничения логической целостности*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$constraint_name	Char(31)	Имя CHECK или NOT NULL ограничения
Rdb$trigger_name	Char(31)	Имя триггера, требующего CHECK-ограничение; для NOT NULL - имя столбца в RDB$RELATION_FIELDS

RDB$COLLATIONS

Содержит данные о порядке (последовательности) сравнения символьных данных. Из таблицы видна связь между допустимыми комбинациями SHAR SET и COLLATION. В частности видно, что для кодовой таблицы Win1251 (ее идентификатор 52) допустимы упорядочения, задаваемые только Winl251 и PXW_CYRL см. также таблицу 7.1. Структура набора символов InterBase.

Таблица 7.3. *Последовательность (порядок) сравнения символьных данньа*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$collation_name	Char(31)	Имя последовательности сравнения
Rdb$collation_id	Smallint	Уникальный идентификатор последовательности сравнения
Rdb$character_set_id	Smallint	Идентификатор кодовой таблицы для последовательности сравнения. Требуется перед выполнением сравнения. Определяет используемую кодовую таблицу. Связан со столбцом RDB$CHARACTER_SET_ID в таблице RDB$CHARACTER_SETS
Rdb$collation_attributes	Smallint	Зарезервировано для внутреннего использования
Rdb$system_flag	Smallint	Указывает, является ли генератор определенным пользователем (значение=0) системным (значение>0)

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$description	BLOB 80	Комментарий пользователя
Rdb$function_name	Char(31)	Зарезервировано для внутреннего использования

RDB$DATABASE

Содержит описание базы данных. Поле Rdb$character_set_name содержит имя кодовой таблицы, используемой по умолчанию.

Таблица 7.4. *Описание базы данных*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$description	BLOB 80	Комментарий пользователя
Rdb$relation_id	Smallint	Зарезервировано для внутреннего использования
Rdb$security_class	Char(31)	Класс секретности, определенный в таблице RDB$SECURITY_CLASSES; ограничения управления доступом, описанные в указанном классе секретности, применяются во всей базе
Rdb$character_set_name	Char(31)	Имя кодовой таблицы

RDB$DEPENDENCIES

Содержит описание зависимостей между объектами базы данных. Для вычисляемых полей, например, одному домену вычисляемого поля соответствует столько строк, сколько аргументов имеет соответствующее выражение.

Таблица 7.5. *Описание зависимостей между объектами базы данных*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$dependent_name	Char(31)	Имя объекта, зависимости (обзор, триггер, вычисляемый столбец)

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$depended_on_name	Char(31)	Имя объекта, аргумента зависимости
Rdb$field_name	Char(31)	Имя столбца объекта, аргумента зависимости
Rdb$dependent_type	Smallint	Тип объекта зависимости: • 0 - таблица • 1 - обзор • 2 - триггер • 3 - вычисляемое поле • 4 - контроль • 5 - процедура • 6 - индексное выражение • 7 - исключение • 8 - пользователь • 9 - поле • 10 – индекс
Rdb$depended_on_type	Smallint	Тип объекта, аргумента зависимости 0-10 (см. выше)

RDB$EXCEPTIONS

Содержит описание исключений в базе данных. Таблица может быть использована, как для выборки текстов исключений, так и для формирования исключений «на лету». Например, в триггере можно сформировать текст исключения, выполнить команду изменения соответствующей строки таблицы RDB$EXCEPTIONS, а затем команду исключения. В результате будет выдано сообщение с измененным текстом. После выдачи исключения происходит откат транзакции, а значит и таблица исключений вернется в прежнее состояние.

Таблица 7.6. *Описание исключений в базе данных*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$exception_name	Char(31)	Имя исключения
Rdb$exception_number	Integer	№ исключения
Rdb$message	Varchar(78)	Текст сообщения исключения

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$description	BLOB 80	Комментарий
Rdb$system_flag	Smallint	Тип исключения: 0 - определенный пользователем >0 - системный

RDB$FIELD_DIMENSIONS

Содержит описание размерностей данных типа массив. Таблица содержит столько строк, сколько полей типа массива описано в базе. В отличие от строк таблицы RDB$EXCEPTIONS вносить в нее изменения нельзя, поскольку эти изменения должны согласовываться с хранимыми в базе данными.

Таблица 7.7. *Описание размерностей массивов*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$field_name	Char(31)	Имя массива (должно быть и в RDB$FIELDS)
Rdb$dimension	Smallint	Указывает на измерение массива (нумерация с 0)
Rdb$lower_bound	Integer	Нижняя граница данного измерения
Rdb$upper_bound	Integer	Верхняя граница данного измерения

RDBSFIELDS

Содержит описание доменов, используемых в базе. Любое поле таблицы обязательно имеет доменное имя. Это либо имя явно описанного домена, либо имя домена, автоматически формируемого при создании таблицы. Таким образом, используя эту таблицу можно получить описание любого поля базы данных.

Таблица 7.8. *Описание полей*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$field_name	Char(31)	Уникальное имя домена или сгенерированного системой имени

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$query_name	Char(31)	Не используется для SQL объектов
Rdb$validation_blr	BLOB 80	Не используется для SQL объектов
Rdb$validation_source	BLOB 80	Не используется для SQL объектов
Rdb$computed_blr	BLOB 80	Для вычисляемых столбцов содержит BLR (Binary Language Representation) выражение, вычисляемое в базе во время выполнения
Rdb$computed_source	BLOB 80	Для вычисляемых столбцов содержит исходное символьное выражение для столбца
Rdb$default_value	BLOB 80	Содержит выражение по умолчанию (BLR)
Rdb$default_source	BLOB 80	Символьное представление значения по умолчанию
Rdb$field_length	Smallint	Длина поля столбца (для несимвольных: 8 - плав. точка двойн., дата, BLOB, Quad; 4 - Long, Float; 2 - Short)
Rdb$field_scale	Smallint	Содержит длину дробной части чисел
Rdb$field_type	Smallint	Задает тип поля: SMALLINT - 7; INTEGER - 8; QUAD - 9; FLOAT - 10; D_FLOAT – 11; CHAR - 14; DOUBLE - 27; DATE - 35; VARCHAR - 37; BLOB - 261
Rdb$field_sub_type	Smallint	Подтип для BLOB: 0 - unspecified; 1 - text; 2 - BLR и т.д.
Rdb$missing_value	BLOB 80	Не используется для SQL объектов
Rdb$missing_source	BLOB 80	Не используется для SQL объектов
Rdb$description	BLOB 80	Содержит пользовательский комментарий
Rdb$system_flag	Smallint	Только для системных таблиц
Rdb$query_header	BLOB 80	Не используется для SQL объектов

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$segment_length	Smallint	Длина сегмента (только для BLOB)
Rdb$edit_string	Varchar(125)	Не используется для SQL объектов
Rdb$external_length	Smallint	Длина поля во внешней таблице, для внутренних - 0
Rdb$external_scale	Smallint	Масштабированная дробная часть (для целых)
Rdb$external_type	Smallint	См. RDB$FIELD_TYPE
Rdb$dimensions	Smallint	Указывает количество измерений для массивов, иначе 0
Rdb$null_flag	Smallint	Пусто - может быть NULL, 1 - не NULL
Rdb$character_length	Smallint	Длина символа в байтах (важно для иероглифов)
Rdb$collation_id		Идентификатор последовательности сравнения
Rdb$character_set_id	Smallint	Кодовая таблица

RDB$FILES

Содержит описание файлов базы данных. Таблица описывает вторичные и теневые файлы базы данных. В тех случаях, когда база реализуется в виде одного файла, данная таблица пуста.

Таблица 7.9. *Описание файлов*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$file_name	Varchar(253)	Имя вторичного или теневого файла базы
Rdb$file_sequence	Smallint	Порядковый номер вторичного или теневого файла базы
Rdb$file_start	Integer	Начальная страница вторичного или теневого файла базы

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$file_length	Integer	Длина файла в блоках
Rdb$file_flags	Smallint	Зарезервировано за системой
Rdb$shadow_number	Smallint	Указывает № набора теневых файлов, иначе 0 (вторичный)

RDB$FILTERS

Содержит описание фильтров BLOB. С каждым фильтром BLOB связана строка таблицы, в которой задается имя фильтра и его характеристики. См. также таблицу RDB$FUNCTIONS, описывающую функции пользователя.

Таблица 7.10. *Описание фильтров BLOB*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$function_name	Char(31)	Уникальное имя фильтра
Rdb$description	BLOB 80	Пользовательский комментарий
Rdb$module_name	Varchar(253)	Имя библиотеки, содержащей исполнимый фильтр
Rdb$entrypoint	Char(31)	Имя точки входа для фильтра BLOB
Rdb$input_sub_type	Smallint	Подтип BLOB для ввода
Rdb$output_sub_type	Smallint	Подтип BLOB для вывода
Rdb$system_flag	Smallint	Для пользовательского фильтра - 0, системного - больше 0

RDB$FORMATS

Содержит описание истории изменения форматов столбцов таблицы. При изменении формата столбца InterBase устанавливает для таблицы новый номер формата. RDB$FORMATS позволяет прикладным программам обращаться к измененным таблицам без перекомпиляции самих программ (описания хранятся в двоичном коде).

Таблица 7.11. *Описание истории изменения форматов таблицы*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$relation_id	Smallint	Определяет таблицу в RDB$RELATIONS
Rdb$format	Smallint	Определяет номер формата таблицы; таблица может иметь любое количество форматов в зависимости от числа обновлений таблицы
Rdb$descriptor	BLOB 80	Содержит список всех столбцов таблицы с указанием их типа, длины и т.д.

RDB$FUNCTION_ARGUMENTS

Содержит описание параметров пользовательских функций (UDF). Каждому параметру UDF соответствует одна строка таблицы.

Таблица 7.12. *Описание параметров пользовательских функций (UDF)*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$function_name	Char(31)	Уникальное имя функции
Rdb$argument_position	Smallint	Номер аргумента в списке аргументов функции
Rdb$mechanism	Smallint	Тип передачи аргумента: 0 - по значению, 1 - по ссылке
Rdb$field_type	Smallint	Тип аргумента (SMALLINT - 7; INTEGER - 8; QUAD - 9; FLOAT - 10; D_FLOAT – 11; CHAR - 14; DOUBLE - 27; DATE - 35; VARCHAR - 37; BLOB - 261)
Rdb$field_scale	Smallint	Масштабный множитель (дробная часть) для данных, представленных целыми
Rdb$field_length	Smallint	Длина параметра (в соответствии с типом)
Rdb$field_subtype	Smallint	Зарезервировано
Rdb$character_set_id	Smallint	Целый идентификатор кодовой таблицы

RDB$FUNCTIONS

Содержит описание пользовательских функций (UDF). Для получения полного описания UDF необходимо также использовать и таблицу RDB$FUNCTION_ARGUMENTS описания параметров UDF. Связать их можно по имени функции: rdb$functions.rdb$function_name = rdb$function_arguments.rdb$function_name.

Таблица 7.13. *Описание пользовательских функций (UDF)*

Имя столбца	***Тип и** длина данных*	***Комментарий***
Rdb$function_name	Char(31)	Уникальное имя функции
Rdb$function_type	Smallint	Зарезервировано
Rdb$query_name	Char(31)	Альтернативное имя функции, которое может быть использовано в isql
Rdb$description	BLOB 80	Пользовательский комментарий
Rdb$module_name	Varchar (253)	Имя библиотеки, содержащей исполнимую функцию
Rdb$entrypoint	Char(31)	Имя точки входа для функции
Rdb$return_argument	Smallint	№ возвращаемого значения в списке аргументов
Rdb$system_flag	Smallint	Для пользовательской функции - 0, для системной – 1

RDB$GENERATORS

Содержит описание генераторов. Помимо пользовательских генераторов содержит и ряд системных, необходимых для генерации уникальных номеров для объектов базы данных.

Таблица 7.14. *Описание генераторов*

***Имя** столбца*	***Тип** и длина данных*	***Комментарий***
Rdb$generator_name	Char(31)	Уникальное имя генератора
Rdb$generator_id	Smallint	Уникальный системный номер генератора

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$system_flag	Smallint	Генератор создан пользователем - 0, системный - 1

RDB$INDEX_SEGMENTS

Содержит описание полей, составляющих индекс. Для каждого индекса содержит столько строк, из скольких полей состоит индекс. Порядок следования полей в индексе задается в столбце Rdb$index_name.

Таблица 7.15. *Описание полей индекса*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$index_name	Char(31)	Имя индекса, частью которого является данный поле
Rdb$field_name	Char(31)	Имя поля
Rdb$field_position	Smallint	№ данного сегмента индекса в соответствии с сортировкой в индексе

RDB$INDICES

Содержит описание индексов таблиц базы данных. Таблица связывает индекс с таблицей, для которой он создается, а также содержит ряд данных, необходимых для оптимизации работы системы.

Таблица 7.16. *Описание индексов таблиц*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$index_name	Char(31)	Имя индекса
Rdb$relation_name	Char(31)	Имя индексируемой таблицы
Rdb$index_id	Smallint	Внутренний идентификатор
Rdb$unique_flag	Smallint	0 - допускает дубликаты, 1 - нет

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$description	BLOB 80	Комментарий пользователя
Rdb$segment_count	Smallint	Количество сегментов в индексе (1 - простой индекс)
Rdb$index_inactive	Smallint	0 - индекс активен, 1 - нет
Rdb$index_type	Smallint	Зарезервировано
Rdb$foreign_key	Char(31)	Имя внешнего ключа, для которого используется индекс
Rdb$system_flag	Smallint	Индекс определен пользователем = 0, системой > 0
Rdb$expression_blr	BLOB 80	Содержит BLR для выражения, вычисляемого СУБД во время выполнения
Rdb$expression_source	BLOB 80	Содержит исходный текст вычисляемого выражения
Rdb$statistics	Double Precision	Коэффициент селективности; используется оптимизатором для формирования стратегии выборки

RDB$LOG_FILES

Не используется.

RDB$PAGES

Хранит историю выделения страниц в базе данных. Может быть использована для анализа интенсивности работы с таблицами базы данных. В основном ориентирована на внутренние нужды.

Таблица 7.17. *Описание истории выделения страниц*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$page_number	Integer	№ физически выделенной страницы
Rdb$relation_id	Smallint	Идентификационный № таблицы, для которой выделена страница

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$page_sequence	Integer	Порядковый № страницы в таблице ранее выделенных страниц
Rdb$page_type	Smallint	Тип страницы (только для внутреннего использования)

RDB$PROCEDURE_PARAMETERS

Содержит описание параметров хранимых процедур. Каждому параметру хранимой процедуры соответствует одна строка таблицы.

Таблица 7.18. *Описание параметров хранимых процедур*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$parameter_name	Char(31)	Имя параметра
Rdb$procedure_name	Char(31)	Имя процедуры
Rdb$parameter_number	Smallint	Порядковый № параметра
Rdb$parameter_type	Smallint	Тип параметра: 0 - входной, 1 - выходной
Rdb$field_source	Char(31)	Глобальное имя столбца
Rdb$description	BLOB 80	Комментарий пользователя
Rdb$system_flag	Smallint	Определен пользователем - 0, системой - больше 0

RDB$PROCEDURES

Содержит описание хранимых процедур. Для получения полного описания хранимой процедуры необходимо также использовать и таблицу RDB$PROCEDURE_PARAMETERS описания параметров хранимых процедур. Связать их можно по имени хранимой процедуры: rdb$procedure_name.rdb$parameter_name = rdb$procedure_parameters.rdb$parameter_name.

См. также RDB$FUNCTIONS - описание пользовательских функций (UDF).

Таблица 7.19. *Описание хранимых процедур*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$procedure_name	Char(31)	Имя процедуры
Rdb$procedure_id	Smallint	№ процедуры
Rdb$procedure_inputs	Smallint	Количество входных параметров
Rdb$ procedure_outputs	Smallint	Количество выходных параметров
Rdb$description	BLOB 80	Комментарий пользователя
Rdb$procedure_source	BLOB 80	Исходный текст кода процедуры
Rdb$procedure__blr	BLOB 80	BLR код процедуры
Rdb$security_class	Char(31)	Класс секретности процедуры
Rdb$owner_name	Char(31)	Имя владельца (создателя) процедуры
Rdb$runtime blob	BLOB 80	Описание метаданных процедуры
Rdb$system_flag	Smallint	Определен пользователем - 0, системой - больше 0

RDB$REF_CONSTRAINTS

Содержит описание ограничений логической целостности данных. В частности здесь описываются все вешние (FOREIGN) ключи.

Таблица 7.20. *Описание ограничений логической целостности данных*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$constraint_name	Char(31)	Имя ограничения
Rdb$const_name_uq	Char(31)	Имя ограничения первичного или уникального ключа
Rdb$match_option	Char(7)	Зарезервировано (по умолчанию - FULL)
Rdb$update_rule	Char(ll)	Задает тип действия с вторичным ключом при изменении первичного (допус-

Имя столбца	*Тип и длина данных*	*Комментарий*
		тимые значения: NO ACTION, CASCADE, SET NULL, SET DEFAULT)
Rdb$delete_rule	Char(11)	Задает тип действия с вторичным ключом при изменении первичного (допустимые значения: NO ACTION, CASCADE, SET NULL, SET DEFAULT)

RDB$RELATION_CONSTRAINTS

Содержит описание ограничений для таблиц. Для каждого ограничения указывается его тип (RIMARY KEY, UNIQUE, FOREIGN KEY, PCHECK, NOT NULL), если ограничение связано с индексом, то обозначение индекса в таблице указывается в поле Rdb$index_name, обеспечивая связь с таблицей RDBSINDICES.

Таблица 7.21. *Описание ограничений для таблиц*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$constraint_name	Char(31)	Имя ограничения
Rdb$constraint_type	Char(11)	Тип ограничения таблицы (RIMARY KEY, UNIQUE, FOREIGN KEY, PCHECK, NOT NULL)
Rdb$relation_name	Char(31)	Имя таблицы
Rdb$deferrable	Char(3)	Зарезервировано (по умолчанию - No)
Rdb$initially_deferred	Char(3)	Зарезервировано (по умолчанию - No)
Rdb$index_name	Char(31)	Имя индекса, используемого ограничениями UNIQUE, PRIMARY KEY, FOREIGN KEY

RDB$RELATION_FIELDS

Содержит описание столбцов таблиц базы данных. По каждому столбцу указана таблица, в которой он используется и доменное имя, таким образом, используя данную таблицу вместе со связанными с ней, можно восстановить полное описание столбца.

Таблица 7.22. *Описание столбцов таблиц базы данных*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$field_name	Char(31)	Имя поля
Rdb$relation_name	Char(31)	Имя таблицы
Rdb$field_source	Char(31)	Имя описания поля в таблице RDBSFIELDS
Rdb$query_name	Char(31)	Альтернативное имя поля для использования в isql; замещает значение в RDBSFIELDS
Rdb$base_field	Char(31)	Только в обзорах; имя столбца из RDBSFIELDS в таблице или обзоре, который является базовым для данного. Для базового столбца: RDB$BASE_FIELD обеспечивает имя столбца, RDB$VIEW_CONTEXT - столбец в этой таблице задает имя исходной таблицы
Rdb$edit_string	Char (125)	Не используется в isql
Rdb$field_position	Smallint	№ столбца в списке (используется в isql для задания порядка вывода столбцов, в gpre - в командах SELECT и INSERT; если несколько столбцов имеют один номер, порядок их вывода не определен)
Rdb$query_header	BLOB 80	Не используется в isql
Rdb$update_flag	Smallint	Не используется в InterBase
Rdb$field_id	Smallint	Идентификатор для использования в BLR для именования столбца
Rdb$view_context	Smallint	Псевдоним, используемый для уточнения столбца обзора, указывая на столбец базовой таблицы; имеет то же значение, что и псевдоним, используемый в BLR
Rdb$description	BLOB 80	Комментарий пользователя
Rdb$default_value	BLOB 80	Описание столбца (BLR)
Rdb$system_flag	Smallint	Определен пользователем - 0, системой - больше 0

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$security_class	Char(31)	Класс секретности в RDB$SECURITY_CLASSES
Rdb$complex_name	Char(31)	Зарезервировано
Rdb$null_flag	Smallint	Определяет, может ли столбец содержать NULL
Rdb$default_source	BLOB 80	Описание столбца (текст)
Rdb$collation_id	Smallint	Идентификатор последовательности сравнения

RDB$RELATIONS

Содержит описание таблиц и обзоров базы данных. Если строка описывает обзор, то поле Rdb$view_source содержится текст команды SELECT, соответствующей обзору. Например, для обзора RUBRICS нашей тестовой базы значением этого поля будет текст:

```
select UNIKEY, BOOKNM from tbook where (matherkey=0)
```

Таблица 7.23. *Описание таблиц и обзоров базы данных*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$view_blr	BLOB 80	Для обзоров. Содержит BLR запроса
Rdb$view_source	BLOB 80	Для обзоров. Содержит текст запроса
Rdb$relation_id	Smallint	Внутренний идентификатор для BLR
Rdb$system_flag	Smallint	Определен пользователем - 0, системой - больше 0
Rdb$dbkey_length	Smallint	Длина ключа базы данных: для таблиц - 8; для обзоров - 8*количество таблиц в обзоре
Rdb$format	Smallint	Только для внутреннего использования InterBase
Rdb$field_id	Smallint	Количество столбцов в таблице
Rdb$relation_name	Smallint	Имя таблицы

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$security_class	Char(31)	Имя класса секретности, определенного в таблице RDB$SECURITY_CLASSES
Rdb$external_file	Varchar (253)	Имя файла, содержащего внешнюю таблицу
Rdb$runtime	BLOB 80	Содержит описание метаданных
Rdb$external_description	BLOB 80	Пользовательское описание внешнего файла
Rdb$owner_name	Char(31)	Имя владельца (создателя) таблицы
Rdb$default_class	Char(31)	Класс секретности по умолчанию
Rdb$flags	Smallint	

RDB$ROLES

Содержит список ролей, определенных в базы данных и их владельцев.

Таблица 7.24. *Описание ролей в базы данных*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$role_name	Char(31)	Имя роли
Rdb$owner_name	Char(31)	Имя владельца (создателя) роли

RDB$SECURITY_CLASSES

Задает список для управления доступом к таблицам, обзорам и их столбцам. По значению поля Rdb$security_class с данной таблицей связаны таблицы описания базы данных - RDB$DATABASE, описание таблиц и обзоров базы данных - RDB$RELATIONS и описание столбцов таблиц базы данных - RDB$RELATION_FIELDS.

Таблица 7.25. *Описание ограничений доступа к объектам базы данных*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$security_class	Char(31)	Имя класса секретности
Rdb$acl	BLOB 80	Список управления доступом, определяющий пользователей, и переданных им прав
Rdb$description	BLOB 80	Комментарий пользователя

RDB$TRANSACTIONS

Хранит историю транзакций, работающих с несколькими базами данных. При работе с одной базой таблица пуста.

Таблица 7.26. *Описание истории транзакций, работающих с несколькими базами данных*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$transaction_id	Integer	Идентификатор multi-database транзакции
Rdb$transaction_state	Smallint	Состояние: 0 - limbo; 1 - committed; 2 - rolled back
Rdb$timestamp	Date	Зарезервировано
Rdb$transaction_description	BLOB 80	Описывает подготовленную multi-database транзакцию, доступную при неудаче повторного соединения

RDB$TRIGGER_MESSAGES

Описывает сообщения системных триггеров с привязкой к конкретному триггеру. Пользовательские триггеры создают свои сообщения, используя механизм исключений, и в этой таблице не появляются.

Таблица 7.27. *Описание сообщения триггеров*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$trigger_name	Char(31)	Имя триггера
Rdb$message_number	Smallint	№сообщения
Rdb$message	Varchar(78)	Текст сообщения

RDB$TRIGGERS

Содержит описание всех триггеров базы данных. Использую текст, хранимый в поле Rdb$trigger_source можно получить исходный текст тела триггера.. Заголовочную часть триггера может быть определена по значениям полей Rdb$trigger_source, Rdb$relation_name, Rdb$trigger_type, Rdb$trigger_inactive, ,смысл которых ясен из приведенной ниже таблицы..

Таблица 7.28. *Описание триггеров*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$trigger_source	Char(31)	Имя триггера
Rdb$relation_name	Char(31)	Имя таблицы
Rdb$trigger_sequence	Smallint	Порядковый № триггера (определяет последовательность выполнения триггеров)
Rdb$trigger_type	Smallint	Тип триггера: 1 - BEFORE INSERT; 2 - AFTER INSERT; 3 - BEFORE UPDATE; 4 - AFTER UPDATE; 5 - BEFORE DELETE; 6 - AFTER DELETE
Rdb$trigger_source	BLOB 80	Текст триггера (исходный)
Rdb$trigger_blr	BLOB 80	Текст триггера (BLR)
Rdb$description	BLOB 80	Комментарий пользователя
Rdb$trigger_inactive	Smallint	0 - триггер активен, 1 - не активен
Rdb$system_flag	Smallint	Определен пользователем - 0, системой - больше 0
Rdb$flags	Smallint	

RDB$TYPES

Содержит перечень типов данных и алиасов символьных наборов и последовательностей сравнения символьных наборов. В версии 5 недоступна. В версии 6 может использоваться. Для каждого типа данных задает возможные его значения, например для типа поля - это TEXT, SHORT, BLOB, CSTRING, TIME и т.д. Для подтипа - ACL, TRANSACTION_DESCRIPTION и т.д. Для механизма передачи данных - BY_VALUE, BY_REFERENCE и т.д. Для типа триггера - PRE_STORE, POST_STORE, PRE_MODIFY, POST_MODIFY и т.д. Для типа объекта - VIEW, TRIGGER, COMPUTED_FIELD и т.д. Для состояния транзакции - LIMBO, COMMITTED, ROLLED_BACK. Для набора символов - NONE, OCTETS, DOS437, WIN1251 и т.д.

Таблица 7.29. *Описание типов данных, кодовых таблиц и последовательностей сравнения*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$field_name	Char(31)	Имя поля, для которого вводится тип
Rdb$type	Smallint	Внутренний код, идентифицирующий тип поля: внутри каждого типа (Rdb$field_name) собственная нумерация, синонимичные наименования (Rdb$type_name) имеют одинаковые коды.
Rdb$type_name	Char(31)	Текст, соответствующий внутреннему коду
Rdb$description	BLOB 80	Комментарий пользователя
Rdb$system_flag	Smallint	Определен пользователем - 0, системой - больше 0

RDB$USER_PRIVILEGES

Содержит сведения о выдачи прав пользователям на основе команд GRANT. Таким образом, таблица может быть использования для восстановления команд выдачи привилегий (GRANT) на использование объектов базы данных.

Таблица 7.30. *Описание прав пользователей*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$user char	Char(31)	Имя пользователя, которому выделены привилегии
Rdb$grantor	Char(31)	Имя пользователя, который выдал привилегии
Rdb$privilege	Char(6)	Привилегия: ALL; SELECT; DELETE; INSERT; UPDATE; REFERENCE; MEMBER OF (for roles)
Rdb$grant_option	Smallint	Привилегия выдана по опции WITH GRANT OPTION - 1 , нет - 0
Rdb$relation_name	Char(31)	Определяет таблицу, для которой дана привилегия
Rdb$field_name char	Char(31)	Для привилегий update - имя поля, для которого дана привилегия
Rdb$user_type	Smallint	
Rdb$object_type	Smallint	

RDB$VIEW_RELATIONS

Описывает обзоры. Для каждого обзора содержит перечень, используемых ими таблиц. Собственно текста SQL для обзора не содержит.

Таблица 7.31. *Описание обзоров*

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$view_name	Char(31)	Имя обзора
Rdb$relation_name	Char(31)	Имя таблицы, используемой в обзоре. Комбинация RDB$VIEW_NAME и RDBSRELATIONNAME должна быть уникальной
Rdb$view_context	Smallint	Код (порядковый номер алиаса), используемый для квалифицирования имен полей

Имя столбца	*Тип и длина данных*	*Комментарий*
Rdb$context_name	Char(31)	Алиас (текстовая версия в SELECT). Эта переменная должна: • Соответствовать значению столбца RDB$VIEW_SOURCE в RDB$RELATIONS • Быть уникальной в обзоре

7.3. Системные обзоры

Используя приведенный ниже SQL script, можно создать четыре обзора, содержащих информацию об ограничениях логической целостности в базе данных. Предварительно, конечно, должна быть создана сама база данных. Системные SQL-обзоры являются подмножеством системных обзоров, определенных стандартом SQL-92. Поскольку они определены в соответствии с ANSI SQL-92, сами имена системных обзоров и их столбцов не начинаются с RDB$.

ОБЗОР CHECK_CONSTRAINTS

```
CREATE VIEW CHECK_CONSTRAINTS (
CONSTRAINT_NAME,
CHECK_CLAUSE
) AS
SELECT RDB$CONSTRAINT_NAME, RDB$TRIGGER_SOURCE
FROM RDB$CHECK_CONSTRAINTS RC, RDB$TRIGGERS RT
WHERE RT.RDB$TRIGGER_NAME = RC.RDB$TRIGGER_NAME;
```

Описывает все CHECK-ограничения, определенные в базе.

Таблица 7.32. *Описание CHECK-ограничений, определенных в базе*

Имя столбца	*Тип и длина данных*	*Описание*
Constraint_name	Char(31)	Имя CHECK ограничения (уникальное)
Check_clause	BLOB 80	Текстовый BLOB. Исходный текст определения триггера (столбец RDB$TRIGGER_SOURCE в таблице rdb$TRIGGERS)

ОБЗОР CONSTRAINTS_COLUMN_USAGE

```
CREATE VIEW CONSTRAINTS_COLUMN_USAGE (
TABLE_NAME,
COLUMN_NAME,
CONSTRAINT_NAME
) AS
SELECT RDB$RELATION_NAME, RDB$FIELD_NAME,
RDB$CONSTRAINT_NAME
FROM RDB$RELATION_CONSTRAINTS RC, RDB$INDEX_SEGMENTS RI
WHERE RI.RDB$INDEX_NAME = RC.RDB$INDEX_NAME;
```

Описывает столбцы, используемые в ограничениях PRIMARY KEY и UNIQUE. Для внешних ограничений (FOREIGN KEY) этот обзор описывает столбцы, определяющие ограничение.

Таблица 7.33. *Описание столбцов, используемых в ограничениях PRIMARY KEY и UNIQUE таблиц базы данных*

Имя столбца	*Тип и длина данных*	*Описание*
Table_name -	Char(31)	Имя таблицы, для которой создано ограничение
Column_name	Char(31)	Имя столбца, используемого в ограничении
Constraint_name	Char(31)	Уникальное имя ограничения

ОБЗОР REFERENTIAL_CONSTRAINTS

```
CREATE VIEW REFERENTIAL_CONSTRAINTS (
CONSTRAINT_NAME,
UNIQUE_CONSTRAINT_NAME,
MATCH_OPTION,
UPDATE_RULE,
DELETE_RULE
) AS
SELECT RDB$CONSTRAINT_NAME, RDB$CONST_NAME_UQ,
RDB$MATCH_OPTION,
RDB$UPDATE_RULE, RDB$DELETE_RULE
FROM RDB$REF_CONSTRAINTS;
```

Описывает все ссылочные ограничения, определенные в базе.

Таблица 7.34. *Описание ссылочных ограничений, определенных в базе*

Имя столбца	***Тип и*** *длина* ***данных***	***Описание***
Constraint_name	Char(31)	Уникальное имя ограничения
Unique_constraint_name	Char(31)	Имя ограничения уникального (UNIQUE) или первичного (PRIMARY KEY) ключа, связанное с указанным списком столбцов
Match_option	Char(7)	Зарезервировано для последующего использования (устанавливается FULL).
Update_rule	Char(ll)	Зарезервировано для последующего использования (устанавливается RESTRICT)
Delete_rule	Char	Зарезервировано для последующего использования (устанавливается RESTRICT)

ОБЗОР TABLE_CONSTRAINTS

```
CREATE VIEW TABLE_CONSTRAINTS (
CONSTRAINT_NAME,
TABLE_NAME,
CONSTRAINT_TYPE,
IS_DEFERRABLE,
INITIALLY_DEFERRED
) AS
SELECT RDB$CONSTRAINT_NAME, RDB$RELATION_NAME,
RDB$CONSTRAINT_TYPE,
RDB$DEFERRABLE, RDB$INITIALLY_DEFERRED
FROM RDB$RELATION_CONSTRAINTS;
```

Описывает все ограничения, определенные в базе для таблиц.

Таблица 7.35. *Описание ограничений, определенных в базе для таблиц*

Имя столбца	***Тип и длина данных***	***Описание***
Constraint_name	Char(31)	Уникальное имя ограничения
Table_name	Char(31)	Имя таблицы, для которой создано ограничение

Имя столбца	*Тип и длина данных*	*Описание*
Constraint_type	Char(31)	Допустимые значения: UNIQUE, PRIMARY KEY, FOREIGN KEY или CHECK
Is_deferrable	Char(3)	Зарезервировано для последующего использования (устанавливается No)
Initially_deferred	Char(3)	Зарезервировано для последующего использования (устанавливается No)

Если для прикладных целей нужно часто использовать данные системных таблиц, то создание собственных обзоров с нужным составом параметров и наименованиями полей является хорошим решением. Такие обзоры хороши также тем, что позволяют защитить данные, собрать их в удобном виде независимо от того в скольких системных таблицах они хранятся. Точно также можно создать и хранимые процедуры.

7.4. Использования описаний данных для прикладных целей

Данные, хранимые в системных таблицах, могут быть использованы в приложениях для получения и обработки информации о составе, структуре и используемых методах контроля и обработки данных в базе, включая и тексты создания объектов на SQL.

Прежде всего, это касается приложений, где пользователю предоставляется в диалоговом режиме формулировать свои запросы к базе данных.

Рассмотрим решение подобных задач на примерах.

Пример 7.1

Выведем список таблиц нашей базы **с** их **описаниями**

```
select Rdb$relation_name, Rdb$description FROM
RDB$RELATIONS
where Rdb$relation_name NOT Like "RDB$%"
and Rdb$view_source is NULL
```

Если воспользоваться пакетом типа QuickDesk, работа с которым будет рассмотрена в главе «Инструментальные средства для работы с InterBase», то можно увидеть таблицу следующего вида.

Таблица 7.36. *Список таблиц тестовой базы по примеру 7.1*

RDB$RELATION_NAME	RDB$DESCRIPTION
TBOOK	Список книг (рубрик)
TAUTHOR	Список авторов
TPLACE	Список места хранения
TREADER	Список читателей
TBOOK1	
TBOOK_AUTHOR	Описание связей авторов и книг
TBOOK_PLACE	Описание связи книга - место
TBOOK_READER	Описание связи книга - читатель
TABLEARR	
E_LIST	
I_LIST	
QD$REPORTS	

Если использовать инструментарий типа WinSQL или IBConsole, то увидеть второй столбец в таком виде не удастся, поскольку в базе это поле хранится, как BLOB.

Поскольку значительная часть описаний, которые нужны для визуализации, хранится именно в так, рассмотрим пример вывода BLOB объектов с короткими текстами в таблицу средствами C++ Builder.

Пример 7.2

Помещаем на форму (Form1) объекты: TDataSource, TQuery, TDBGrid (DataSource1, Query1, DBGrid1).

Устанавливаем свойство объекта Form1 Caption = "Список таблиц".

Устанавливаем свойство объекта DataSource1 DataSet = Query1.

Устанавливаем свойство объекта DBGrid1 DataSource = DataSource1.

Устанавливаем свойство объекта Query1 AutoCalcFields = true.

Устанавливаем свойство объекта Query1 DataBase = TESTLIBR (алиас нашей тестовой базы).

Устанавливаем свойство объекта Query1 SQL = «select Rdb$relation_name, Rdb$description FROM RDB$RELATIONS

where Rdb$relation_name NOT Like "RDB$%"

and Rdb$view_source is NULL»

Вызываем FieldEditor, добавляем в него все выбираемые поля, а также добавляем новое вычисляемое поле DESCRIPTION, как строковое данное длиной 100 байтов.

Выбираем событие OnCalcFields. После двойного «клика» у нас создается заголовок обработчика событий Query1CalcFields.

Теперь набираем текст обработки события, содержащий текст преобразования BLOB в символьную строку. При этом в строке нужно убрать коды «возврат каретки» и «перевод строки».

Текст будет иметь вид

```
void__fastcall TForm1::Query1CalcFields(TDataSet *DataSet)
{
 TBlobField * tt=
  dynamic_cast<TBlobField *>
 (Query1->FieldByName("Rdb$description"));
 TStringList *TS=new TStringList;
// Настраиваемся на работу с BLOB
 TBlobStream *bs = new TBlobStream(tt, bmRead);
// Создаем поток для чтения из выбранного BLOB
 TS->LoadFromStream(bs);
// Записываем данные BLOB в объект TStringList
// (список строк)
 AnsiString SText=TS->Text;
 for(int i=1;i<=SText.Length();i++)
  if(SText[i]==13 || SText[i]==10) SText[i]=' ';
// Убираем коды «возврат каретки» и «перевод строки».
 Query1->FieldByName("description")->AsString=SText;
// «Записываем» в Query
 delete bs; // удаляем поток
}
```

В ColumnEditor для объекта DBGridl задаем заголовки столбцов «Имя таблицы», «Описание таблицы», удалив предварительно столбец RDB$DESCRIPTION.

Устанавливаем свойство объекта Query1 Active = true и запускаем приложение на выполнение. В режиме проектирования вычисляемые поля, в данном случае «Описание таблицы», не видны, так как для их заполнения необходимо выполнить записанный нами код.

В результате получим картинку, представленную на рисунке 7.1.

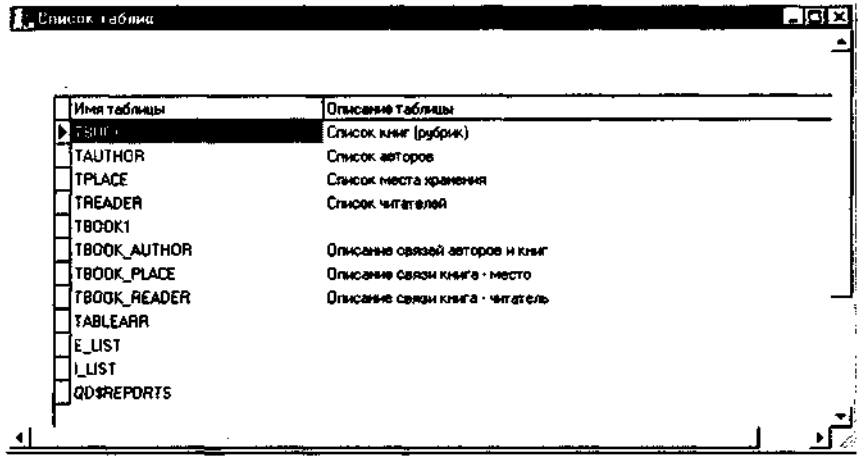


Рис. 7.1. Представление данных с BLOB в табличном виде

Такой подход удобен, если нужно однократно в какой-либо форме представить данные BLOB. Если же данные BLOB удобно регулярно преобразовывать в символьные строки, то можно написать специальную UDF, выполняющую подобные действия.

Пример 7.3

Для преобразования BLOB в строку создадим UDF blob_str. Текст ее может быть, например, следующим.

```
char * __declspec(dllexport)  blob_str(SBLOB Ы)
{
 long length=100;
 char *buffer,t,*retstr;
 int curpos=0;
 retstr=(char *)malloc(256);
 retstr[0]=0;
 if(!(bl->blob_get_segment)) return retstr;
 buffer=new char[bl->max_seglen+1];
 for(int i=0;i<bl->number_segments && curpos<255;i++)
 {(bl->blob_get_segment)(bl->blob_handle,buffer,
    bl->max_seglen,length);
  for(int i=0;i<length && curpos<255;i++,curpos++)
  {t=buffer[i];
   if(t==10 && t==13) t=' •;
   retstr[curpos]=t;
  }
 }
```

```
 retstr[curpos]=0;
 delete!] buffer;
 return retstr;
}
```

Описание UDF в базе тогда будет:

```
DECLARE EXTERNAL FUNCTION BLOB_STR
    BLOB
    RETURNS CSTRING(255) FREE_IT
    ENTRY_POINT '_blob_str' MODULE_NAME 'MYDLL';
```

Пример 7.4

После создания подобной UDF наша выборка будет выглядеть

```
select Rdb$relation_name, BLOB_STR(Rdb$description)
 FROM RDB$RELATIONS
 where Rdb$relation_name NOT Like "RDB$%"
   and Rdb$view_source is NULL
```

Поскольку в выбираемых полях уже нет BLOB, то такая выборка будет пригодна как для интерактивных утилит, так и для любых приложений. Нужно только помнить, что она обрезает хранимый текст до 255 байт.

Результат такой выборки уже показан в таблице 7.36.

Рассмотрим теперь выборку списка полей из таблицы вместе с их описаниями.

Пример 7.4

```
select Rdb$field_position+1 NUMBER,
  Rdb$field_name Name,
  BLOB_STR(Rdb$description) Description
 FROM RDB$RELATION_FIELDS
 where Rdb$relation_name = "TBOOK"
```

Теперь добавим к ним сведения о типе полей, их длине и точности.

```
select a.Rdb$field_position+1 NUMBER,
  a.Rdb$field_name Name,
  b.Rdb$field_type FieldType,
  b.Rdb$field_length FieldLength,
  Rdb$field_scale Scale,
  BLOB_STR(a.Rdb$description) Description
```

Аналогичным образом непосредственно или с соответствующей обработкой в программе можно получить описание, как для наглядного представления, так и для получения собственно команд SQL, практически всех объектов базы данных.

И, наконец, рассмотрим возможность создание собственных исключений непосредственно во время работы триггеров (хранимых процедур). Последнее имеет смысл, прежде всего тогда, когда стандартный текст должен дополняться теми или иными дополнениями. В качестве примера используем текст триггера I_TBOOK_1. Его тело, собственно уже было приведено в предыдущем примере. Дадим теперь его полностью.

Пример 7.5

```
CREATE TRIGGER I_TBOOK_1 FOR TBOOK
ACTIVE BEFORE INSERT POSITION 0
as
begin
  if (new.UNIKEY is NULL) then
   new.UNIKEY=GEN_ID(sysnumber,1);
  if (new.MATHERKEY is NULL or new.MATHERKEY<0) then
  BEGIN
   update RDB$EXCEPTIONS SET
    Rdb$message='Не указана рубрика для <'
          || new.BOOKNM || '>'
    where Rdb$exception_name='NO_RUBRIC';
   exception NO_RUBRIC;
  END
  if (new.MATHERKEY>0) then
   if (NOT EXISTS (select * from TBOOK
     where (unikey=new.MATHERKEY)))
   then exception ERR_RUBRIC;
  if (new.BOOKNM is NULL) then exception NO_BOOKNM;
end
```

В данном примере при недопустимом значении поля MATHERKEY формируется исключение, но с той особенностью, что в текст исключения добавляется наименование книги - поле BOOKNM. Для этого сначала изменяется строка таблицы исключений, а затем выдается само исключение (уже измененное). Его текст и получит конечный пользователь. Далее выдача исключения вызывает откат транзакции, в результате все изменения, сделанные транзакцией, следовательно, и изменения в таблице исключений будут отменены, а само исключение сохранится в первоначальном виде, то есть как раз то, что нам и нужно.

Такой механизм предполагает внесение и откат «лишних» изменений, что, конечно, вызывает замедление обработки, поэтому не стоит им злоупотреблять, но в ряде случаев он может быть весьма полезен.

Глава 8

Администрирование базы данных

Начнем с замечания, звучащего несколько рекламно, но, тем не менее, полностью соответствующего действительности. Большинство SQL-серверов требуют целых подразделений, занимающихся только обслуживанием SQL-сервера, его настройкой и управлением. InterBase обеспечивает не только высокую производительность использующих его систем, но и простоту их сопровождения, предоставляя возможность создания баз данных любого уровня - от персональных до корпоративных с сотнями пользователей.

Архитектура InterBase эффективно использует ресурсы системы. Для установки достаточно 10Мб на диске (большую часть занимают справочные файлы и примеры программирования) и минимальное количество оперативной памяти, достаточное для работы операционной системы, что выгодно отличает InterBase от большинства других продуктов, которые требуют существенно большего количества памяти и серверных ресурсов.

В то же время не следует считать, что проблем администрирования базы не существует. Эффективность работы системы во многом зависит от того, насколько эффективно спроектирована база, как поддерживается ее целостность. Как решаются проблемы распределения доступа к данным, что, собственно и составляет основную задачу администратора.

Масштаб этих задач может изменяться от регулярного копирования данных, обеспечивающего их сохранность в небольших системах, до достаточно сложных операций по контролю за доступом к данным, изменениям в их структуре и их оптимизации в многопользовательских, особенно распределенных системах.

Здесь рассматриваются только основные задачи. Мы оставляем в стороне проблемы, возникающие в крупных системах.

8.1. Установка InterBase

IB Database устанавливается запуском setup с дистрибутива. После запуска установки в Windows выводится картинка, содержащая перечень компонент, включенных в дистрибутив, и предлагаемую директорию для установки.

Указываем требуемую директорию, если предлагаемая не устраивает, и помечаем компоненты, которые хотим включить в установку. Практически можно рекомендовать полную установку, учитывая, что занимаемый объем не превышает 30 Мб (вместе с Adobe Acrobat Reader - 36), из них около 20 занимают документация и примеры.

После ответов на вопросы о каталоге установки и устанавливаемых компонентах появляется картинка-заставка процесса установки InterBase.

В процессе установки необходимые файлы переписываются с дистрибутива на винчестер и настраиваются. Вся процедура установки занимает не более пяти минут.

IB Database динамически настраивается на количество дисковой и оперативной памяти или на количество работающих пользователей, поэтому нет необходимости настраивать сервер для получения максимальной производительности.

При установке систем для нескольких пользователей достаточно установить один комплект документации, во всяком случае, не более одного комплекта на каждого пользователя-разработчика.

Установка клиента осуществляется аналогично установке сервера, только задается меньше вопросов и заканчивается быстрее. Для размещения клиентской части требуется чуть более 2 Мб дисковой памяти.

Установка на платформах, отличных от Windows / Windows NT, имеет некоторую специфику, что отражено в соответствующей документации, но также не займет более 15 минут.

В целом можно сказать, что установка InterBase больше похожа на установку программ в среде DOS, а не Windows NT. Правда, надо ясно понимать, что установить InterBase и научиться эффективно использовать его возможности, это не одно и то же.

Чтобы настроить сервер, запускаем утилиту InterBase Configuration (regcfg.exe) и выбираем желаемые режимы запуска (рис. 8.1).

Настройка и обслуживание базы с помощью диспетчера серверов

Настройка базы данных и ее обслуживание, включая резервное копирование и восстановление базы данных, осуществляется другой утили-

той - диспетчером серверов: InterBase Server Manager (ibmgr32.exe).* Его окно представлено на рис. 8.2.

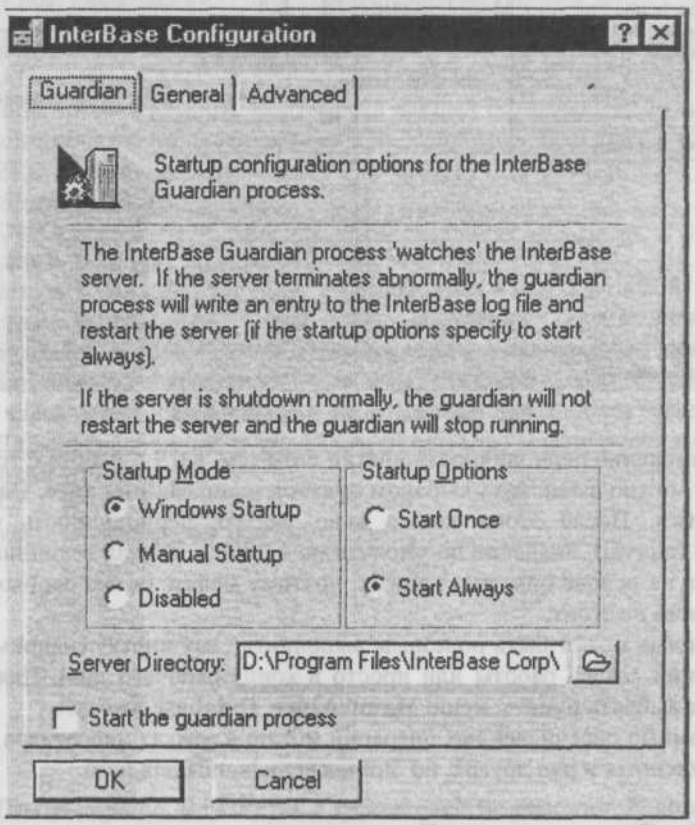


Рис. 8.1. Задание конфигурации сервера InterBase.

В рамках диспетчера серверов настраивается список пользователей, их пароли и т.п. Поэкспериментируйте, это не сложно и, пока базы данных не заполнены, абсолютно безопасно. Главное, это не забыть пароли.

Оставаясь в диспетчере серверов, можно выполнить копирование базы данных выбором пунктов меню Tasks, Backup. Восстановление базы по копии - выбором пунктов меню Tasks, Restore.

* В версии InterBase 6 для реализации этих функций используется утилита IBConsole.exe, интегрирующая ряд утилит предыдущих версий.

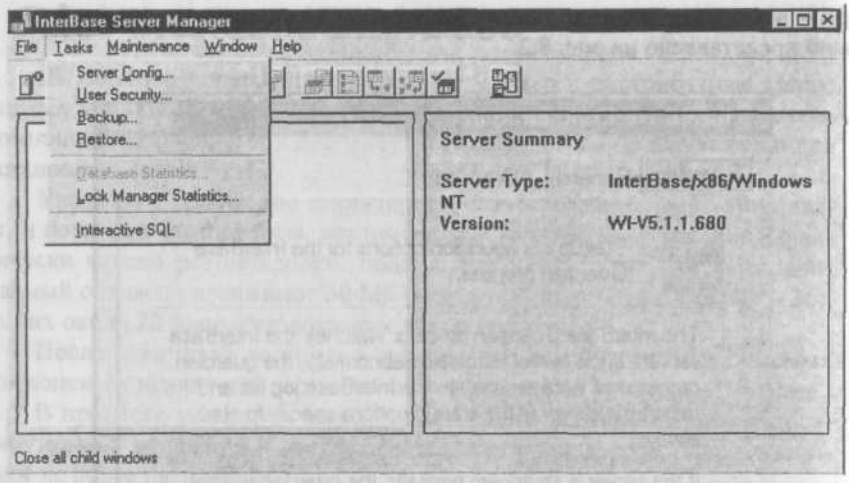


Рис. 8.2. Окно диспетчера серверов.

Изменение периодичности чистки базы (см. гл. 9 о работе с транзакциями) можно выполнить выбором пунктов меню **Maintenance, Database Properties.** После этого можно явно указать периодичность чистки (Sweep Interval). Значение по умолчанию - 20000. Данная величина установлена на основе опытных данных, поэтому менять ее без особых на то оснований не стоит.

Иногда может быть полезным выполнить саму чистку (например, по окончании сеанса работы или просто в конце рабочего дня). Для этого следует выбрать пункты меню **Maintenance, Database Sweep.**

Помимо перечисленных операций можно в рамках диспетчера серверов выполнить и ряд других, но они неспецифичны для него.

8.2. Создание базы данных

Создание базы данных удобнее всего произвести, используя либо WinSQL в InterBase 5, либо утилиту IBConsole в InterBase 6 с переходом в ней в Interactive SQL.

Далее выполняем команду CREATE DATABASE.

В качестве примера возьмем создание нашей тестовой базы на локальной машине. Существенное значение при этом имеет задание кодовой таблицы, используемой по умолчанию для данных, хранимых в базе. Для хранения данных на русском языке пригодны два варианта. Рассмотрим преимущества и недостатки каждого из них.

Создание базы данных без кодовой таблицы.

Пример 8.1

```
CREATE DATABASE "C:/MYDIR/TESTBASE.GDB" USER "SYSDBA"
PASSWORD "masterkey";
```

В этом случае символьные данные хранятся в базе в том виде (DEFAULT CHARACTER SET NONE), как они были загружены, без каких-либо преобразований. Сортировка данных осуществляется в порядке возрастания кодов хранимых символов. Например, если столбец таблицы C_FIELD таблицы ABC содержит значения F, f, g, G, Ц, ц, Б, б, то в результате сортировки

SELECT C_FIELD FROM ABC ORDER BY C_FIELD

получим F, G, f, g, Б, Ц, б, ц.

Задать COLLATE, нельзя, поскольку для каждой кодовой таблицы допустим строго определенный набор допустимых упорядочений. Для NONE их нет вообще. Таким образом, порядок сортировки здесь фиксирован именно по возрастанию кодов и изменить его нельзя. С другой стороны здесь нет преобразований и такая сортировка наиболее эффективна.

Для сортировки вне зависимости от регистра можно воспользоваться функцией UPPER.

SELECT C_FIELD FROM ABC, UPPER(ABC) ORDER BY 2

в результате сортировки получим:

F	F
f	F
g	G
G	G
Б	**Б**
Ц	Ц
б	б
ц	ц

Почти хорошо, но UPPER с кириллицей не работает. К сожалению, этот факт не зависит от используемой кодовой таблицы. Однако проблема эта легко решается с помощью подключения соответствующей UDF. В приложении приведен текст такой функции на C и ее объявление в базе. Функция названа RUPPER. Воспользуемся ей.

SELECT C_FIELD FROM ABC, RUPPER(ABC) ORDER BY 2

в результате сортировки получим:

F	F
f	F
g	G
G	G
Б	Б
б	Б
Ц	Ц
ц	Ц

Следует отметить, что функция типа RUPPER нужна не только для сортировки, но и для сравнения данных, например в условиях, если необходимо устранить зависимость результата от регистра.

Создание базы данных с кодовой таблицей WIN1251.

Пример 8.2

```
CREATE DATABASE "C:/MYDIR/TESTBASE.GDB" USER "SYSDBA"
PASSWORD "masterkey" DEFAULT CHARACTER SET WIN1251;
```

Значение для упорядочения (COLLATE) при этом будет также WIN1251. Сортировка данных осуществляется в порядке возрастания кодов хранимых символов, если при описании данных (доменов) не указана конструкция COLLATE. Например, если столбец таблицы C_FIELD таблицы ABC содержит значения F, f, g, G, Ц, ц, Б, б, то в результате сортировки

```
SELECT C_FIELD FROM ABC ORDER BY C_FIELD
```

получим F, G, f, g, Б, Ц, б, ц.

Здесь, правда можно задать COLLATE, как при описании доменов или описании столбцов таблиц (это, по существу, одно и то же, поскольку описание столбца в таблице, если он явно не ссылается на домен, порождает генерируемый системой домен), либо непосредственно при выборке.

Рассмотрим соответствующий пример

```
SELECT C_FIELD FROM ABC ORDER BY C_FIELD COLLATE PXW_CYRL
```

в результате сортировки получим: f, F, g, G, б, Б, ц, Ц.

То есть в данном случае мы можем выполнять сортировку в двух режимах.

Еще раз отметим, что функция UPPER и в этом случае не будет работать для кириллицы, так что от UDF функций типа RUPPER все равно не уйти.

При создании базы можно также указать такие параметры, как размер страницы (PAGE_SIZE), вторичные файлы и их характеристики. Последние существенны для оптимизации работы больших баз. На первых порах лучше использовать применяемые по умолчанию значения, тем более что необходимые изменения можно внести и в дальнейшем.

Полный синтаксис команды CREATE DATABASE приведен в приложении.

8.3. Настройка BDE

Назначение BDE и организация связи с ним приложения

Проблемы настройки BDE (Borland Database Engine) при работе с InterBase возникают, прежде всего, в системах, работающих с C++Builder и Delphi. Последние, однако, являются, пожалуй, наиболее распространенными средствами для разработки систем, работающих с СУБД, поэтому этот вопрос представляется достаточно важным.

Существует несколько способов организации интерфейса с базами данных. Выделим два основных.

Первый способ - это работа средствами API соответствующей СУБД. При этом способе обеспечивается максимальное быстродействие приложения. Платой за это является жесткая привязка системы к конкретной СУБД и, как следствие, непереносимость системы. Для перехода на работу с другой СУБД требуется переработка программного обеспечения приложения.

Второй способ - это работа с использованием средств пакета, обеспечивающего настройку на работу с конкретной СУБД. При этом способе доступ к базе осуществляется в несколько этапов, что приводит в определенной степени к снижению эффективности работы приложения. Компенсацией за это является независимость приложения от конкретной СУБД, Это обеспечивает возможность простоты перехода от одной СУБД к другой. Кроме того, появляется возможность использовать в работе большой набор стандартных компонент для работы с базами данных, что ускоряет разработку приложений.

Выбор конкретного способа - за разработчиком программного обеспечения. Если разработка ведется силами программистов конкретной фирмы для нужд этой фирмы и в обозримом будущем никаких изменений в выбранном программном обеспечении не предполагается (что, вообще говоря, трудно гарантировать), то первый способ предпочтительней. Если же разработка ведется для различных фирм, то второй способ явно предпочтительней, поскольку в этом случае удается избежать многих проблем при переходе с одной платформы на другую. В противном случае придется отказываться от ряда заказов, что едва ли оправдано.

Рассмотрим подробнее реализацию разработки вторым способом на основе использования средств BDE.

Схема взаимодействия приложения с базой данных приведена на рис. 8.3.

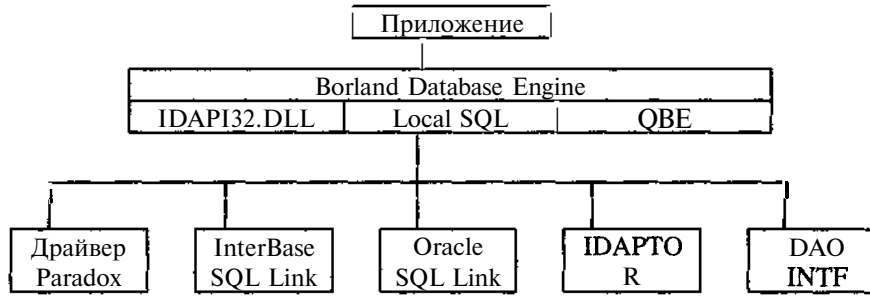


Рис. 8.3. Взаимодействие приложения с базой данных.

Поскольку приложение связано только с BDE, то оно явно не зависит от конкретной СУБД, однако BDE должно знать, с чем именно оно должно связываться. Следовательно, для BDE необходимо указать идентификатор конкретной базы. С каждым таким идентификатором связывается описание соответствующей базы. После того как все необходимые описания созданы, дальнейшая забота по работе с базой может быть поручена BDE.

Итак, рассмотрим задачу настройки описаний базы данных для BDE.

Для идентификации базы данных используется ее символьный идентификатор - алиас базы данных. Алиас известен приложению и с алиасом связано описание, используемое BDE.

С каждым алиасом необходимо связать:

- тип базы данных;
- фактическое имя и путь доступа к базе;
- дополнительные характеристики базы, необходимые для настройки на работу с ней.

Настройка BDE для работы с базой InterBase (использование BDE Administrator)

Наиболее удобным и естественным способом настройки BDE для работы с конкретным приложением является использование утилиты BDE Administrator (bdeadmin.exe).

Стартуем BDE Administrator. Получаем окно, показанное на рис. 8.4.

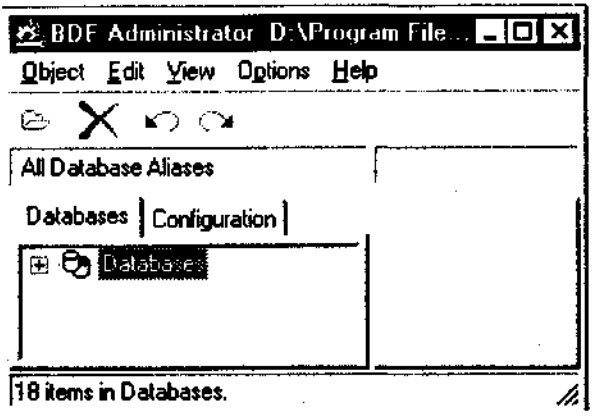


Рис. 8.4. Главное окно администратора баз данных.

Выбираем пункт меню Object, подпункт New (то же самое можно сделать и другими способами, но, чтобы не загромождать описание, ограничимся одним).

Получаем окно с заголовком New Database Alias. В нем выбираем из предлагаемого списка драйверов (Database Driver Name) драйвер для InterBase - INTRBASE. Нажимаем OK. Получаем новое окно (рис. 8.5).

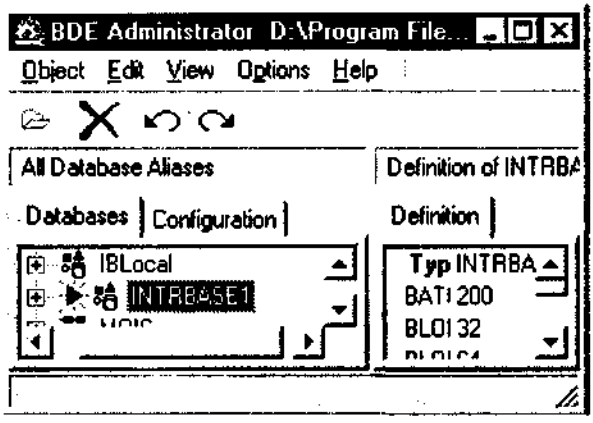


Рис. 8.5. Окно настройки баз данных администратора баз.

Вводим имя для алиаса, в нашем случае - вместо INTRBASE1. Например, TESTLIBR. В окне справа вводим в поле SERVER NAME фактический путь доступа к базе данных. В поле USER NAME указываем имя

пользователя. Нажимаем на голубую стрелку - Apply. Описание базы создано.

Теперь можно открыть базу. Для этого щелкаем мышью на знаке '+' перед именем алиаса. Вводим имя пользователя и пароль. Соединение с базой выполнено. Еще раз выберем пункт меню Object.

Теперь можно перейти к работе с базой, вызвав, например, диспетчер серверов - пункт меню Server Manager или утилиту для работы с базой данных WinSQL для InterBase 5 или IBConsole для InterBase 6, далее пункт меню для работы с ISQL.

На этом, собственно, заканчивается создание алиаса. Приложение уже может работать с базой данных, используя алиас.

Настройка BDE на работу с кириллицей

Теперь несколько слов о настройке BDE для поддержки работы с кириллицей.

В процессе работы средства BDE читают или пишут данные из базы. При этом при необходимости производится перекодирование данных.

Внутри Windows-приложения символьные данные хранятся в текущей кодировке, для русскоязычных приложений - 1251.

Итак, если мы имеем, например файлы DBF в кодировке DOS 866, то для их штатного прочтения необходимо ее задать. Для этого надо войти в BDE Administrator в подменю Configuration, выбрать Native, DBASE и задать LANGDRIVER dBASE RUS cp866. В этом случае производится перекодирование данных как при чтении, так и при записи, таким образом можно «забыть», что на диске данные хранятся в кодировке DOS, а в приложении - в кодировке Windows. Но в данном случае сами файлы не содержат информации об используемой кодовой странице.

При работе с InterBase ситуация усложняется тем, что и внутри InterBase хранится информация об используемой кодовой странице. Рассмотрим следующие варианты и возникающие в них проблемы.

В InterBase задано CHARACTER SET NONE

Здесь возможны 2 случая.

Вариант 1.

В BDE не указан LANGDRIVER (поле пусто).

В этом случае перекодировка данных при чтении и записи не производится. При работе с InterBase никаких проблем не возникает.

Теперь читаем данные из файла DBASE (LANGDRIVER dBASE RUS cp866) в приложение. Данные перекодируются. Далее из приложения записываем построчно в InterBase (NONE). Перекодировки не происходит. Записано все правильно и в нужной нам кодировке.

Теперь проделываем то же самое, но в один прием, используя локальный SQL.

Пусть файл FILET.DBF в директории E:\userdos содержит два символьных поля: TXT1 и TXT2.

Создадим в нашей базе таблицу TABL1.

```
CREATE TABLE SPLAV (
 TXT1 VARCHAR(5),
 TXT2 VARCHAR(45));
```

Тогда данные из файла DBASE - FILET.DBF в нашу таблицу можно поместить с помощью средств локального SQL BDE в нашу базу следующей командой.

```
insert into ":TESTLIBR:TABL1"(TXT1, TXT2)
select cast(TXTl as varchar(5)), cast(TXT2 as varchar(45))
 from 'E:\userdos\FILET.DBF'
```

Промежуточного перекодирования теперь нет, данные попали в базу, но вместо кириллицы появились #.

Таким образом, прямая запись в данной ситуации невозможна.

То же можно сказать и о выгрузке данных из базы.

Вариант 2.

В BDE указан LANGDRIVER Pdox ANSI Cyrillic. Это соответствует кодировка WIN1251.

При обращении к базе получаем сообщение:

General SQL error.

arithmetic exeption, numeric overflow, or string truncation

Cannot transliterate character between character sets.

Проще говоря, этот вариант непригоден. BDE не умеет выполнять перекодировки с NONE.

В InterBase задано CHARACTER SET WIN1251.

Здесь возможны 2 случая.

Вариант 1.

В BDE не указан LANGDRIVER (поле пусто).

Проведем те же манипуляции, что и раньше.

Чтение из базы прошло, поскольку при чтении перекодировки нет. Запись идет из NONE в WIN1251 - BDE этого не умеет.

Прямая запись также не прошла. Здесь перекодировки DOS 866 - NONE - WIN1251 результат тот же, что и при установке в базе кодовой таблицы NONE. Этот вариант также непригоден.

Вариант 2.

В BDE указан LANGDRIVER Pdox ANSI Cyrillic. Это соответствует кодировка WIN1251.

Чтение из базы прошло, поскольку кодовые страницы совместимы.

Записи, как в построчном режиме, так и с помощью SELECT прошли правильно. DOS 866 - Pdox ANSI Cyrillic - WIN1251.

Возникают ли здесь какие-либо проблемы. Увы, да. Если посмотреть системные таблицы, то можно увидеть, что в них явно указана кодовая таблица character set UNICODE_FSS. Для приложений, это не беда, но, если нужно пользоваться утилитой SQL Explorer, то при попытке просмотра списка таблиц базы он выдаст сообщение, которое мы уже видели: *Cannot transliterate character between character sets.* Связано это с тем, что для получения этого списка SQL Explorer пытается читать системные таблицы, а они как раз «не в той кодировке». Беда, конечно не так велика, но к ней нужно быть готовым.

Подведем итоги.

Получить возможность работы с русскими буквами в базе данных можно двумя способами.

Первый способ является более простым. При создании базы данных DEFAULT CHARATER SET ни для базы данных, ни для символьных полей не указывается. В BDE в поле параметра LANGDRIVER задаем значение пусто. В этом случае в БД можно записывать символы в любой кодировке.

При таком способе единственной проблемой при работе с базой данных является то, что при сортировке данных сначала пойдут прописные буквы, а потом строчные. Для сортировки вне зависимости от регистра нужна UDF типа RUPPER. Занесение данных из внешних таблиц следует делать построчно.

Другой способ предполагает задание при создании БД дополнительного параметра DEFAULT CHARACTER SET WIN1251.

При работе с использованием BDE необходимо выполнить его настройку. Удобнее всего воспользоваться утилитой администратора баз данных (bdeadmin.exe), описанной выше. Для этого достаточно на страничке System для драйвера INTRBASE (чтобы потом не делать то же самое каждый раз при создании нового псевдонима) или для псевдонимов INTRBASE установить параметр LANGUAGE DRIVER = **Pdox ANSI Cyrillic.**

При установке кодировки WIN1251 collation устанавливается по умолчанию также в WIN1251, сортировка текстов (так же, как и при первом способе) будет выполняться по возрастанию кодов символов. Для обеспечения сортировки независимо от регистра необходимо при описании доменов (полей таблиц) или непосредственно в конструкции команды SELECT после имени поля в ORDER BY указать collate PXW_CYRL.

Рассмотрим результаты сортировок на примерах.

Пример 8.1

```
CREATE TABLE TAUTH0R1  (
    AUTHOR  PRMKEY,
    AUNAME VARCHAR(60)  COLLATE PXW_CYRL,
    COMMENT  VARCHAR(80),
);
```

Зададим следующие значения.

Таблица. 8.1. *Содержимое таблицы TAUTHOR1 для примера 8.1*

AUTHOR	*AUNAME*	*COMMENT*
19	Дашкова Полина	Первый
20	Хмелевская Иоанна	второй
21	Ладыжинская Ольга Александровна	Третий
22	Бурова И.И.	четвертый
24	без авторов	Пятый

Применим команду:

Пример 8.2

```
select * from tauthor order by AUNAME;
```

Результат представлен в табл. 8.2.

Таблица 8.2. *Результат выборки с сортировкой при наличии опции COLLATE*

AUTHOR	*AUNAME*	*COMMENT*
24	без авторов	Пятый
22	Бурова И.И.	четвертый
19	Дашкова Полина	Первый
21	Ладыжинская Ольга Александровна	Третий
20	Хмелевская Иоанна	Второй

Сортировка выполнена по 2 столбцу, причем регистр текста не учитывается.

Теперь выполним такую же операцию, но по третьему столбцу, по которому COLLATE не задана.

Пример 8.3

```
select * from tauthor order by COMMENT;
```

Таблица 8.3. *Результат выборки с сортировкой при отсутствии опции COLLATE*

AUTHOR	*AUNAME*	*COMMENT*
19	Дашкова Полина	Первый
24	без авторов	Пятый
21	Ладыжинская Ольга Александровна	Третий
20	Хмелевская Иоанна	второй
22	Бурова И.И.	четвертый

Здесь также выполнена сортировка, но уже только по возрастанию кодов символов, а это означает, что вначале идут прописные буквы, а уже потом строчные.

Чтобы провести сортировку, аналогичную первой, достаточно сделать следующее.

Пример 8.4

```
select * from tauthor order by COMMENT COLLATE PXW_CYRL;
```

Таблица 8.4. *Результат выборки с сортировкой с опцией COLLATE в запросе*

AUTHOR	*AUNAME*	*COMMENT*
20	Хмелевская Иоанна	Второй
19	Дашкова Полина	Первый
24	без авторов	Пятый
21	Ладыжинская Ольга Александровна	Третий
22	Бурова И.И.	четвертый

В списке кодировок Borland InterBase есть еще одна русскоязычная - CYRL (866), соответствующая кодировке в DOS, однако ее лучше не использовать, поскольку это немедленно приведет либо к проблеме со шрифтами в приложении, либо к необходимости в постоянной перекодировке данных. Тем более что какой-либо необходимости в этом не видно, разве что при использовании старых баз, созданных в DOS. Но и в этом случае проще перекодировать их при перемещении в InterBase.

Следует, правда, отметить, что собственно утилиты BDE не всегда корректно ведут себя с упомянутыми кодировками, особенно, если необходимо одновременно работать с базами InterBase и локальными таблицами Paradox или dBASE. В этом смысле работа без явного указания CHARACTER SET предпочтительнее. На сегодняшний день здесь не возникает никаких проблем.

8.4. Управление доступом к данным

Управление доступом к данным в системах с одним или несколькими пользователями не представляет проблемы. Все данные доступны для всех. Если пользователей у системы много, то возникает проблема защиты данных от несанкционированного доступа, чаще всего это связано не столько с секретностью данных, хотя в ряде случаев они могут носить и сугубо конфиденциальный характер, сколько с защитой, как от случайных, так и преднамеренных искажений.

Для этого формируется список пользователей и каждому пользователю предоставляются права на доступ к определенным данным.

Управление доступом к данным включает, прежде всего, управление списком пользователей и правами их доступа к данным и процедурам.

При большом числе пользователей следить за предоставлением прав каждому пользователю становится малоприятной задачей, тем более что во многих случаях пользователи явно распадаются на группы, например кассиры в большом магазине. Ясно, что все они работают с одними и теми же данными, одними и теми же средствами, следовательно, должны иметь те же права, а имя пользователя нужно только для идентификации вносимых ими данных.

Для регулирования доступа таких групп в InterBase предусмотрен механизм «ролей».

Рассмотрим подробнее процедуры управления списками пользователей и ролей, предоставления им прав доступа к объектам базы данных.

Создание списка пользователей

Рассмотрим выполнение этой работы в среде Windows. Для создания списка пользователей можно воспользоваться диспетчером серверов - InterBase Server Manager (ibmgr32.exe) при работе с InterBase версий 4 и 5, либо IBConsole.exe для версий 6.

В первом случае стартуем диспетчер серверов.

Соединяемся с конкретной базой. Выбираем пункт меню **File** и внутри него пункт **Server Login.** Вводим пароль. Если данные введены правильно, то происходит соединение с базой. Для обеспечения работы по созданию пользователей необходимо иметь права администратора базы данных. При первом соединении: пользователь - **SYSDBA,** пароль - **masterkey.**

Далее выбираем пункт меню Tasks и внутри него пункт **User Security.** Добавляем нового пользователя, выбрав **Add User.** Указываем имя пользователя (User Name), используемое для его идентификации, и пароль. При желании можно задать дополнительные данные о пользователе: фамилию, имя.

Во втором случае стартуем IBConsole.

Входим в режим регистрации сервера. Аналогично первому случаю вводим имя пользователя и пароль.

Открываем пункт Users и в диалоговом режиме добавляем пользователя или меняем его характеристики.

Пользователь создан. При необходимости данные пользователя можно модифицировать или удалить. Единственное, что не рекомендуется, это удаление пользователя SYSDBA. Последнее связано с необходимостью переустановки InterBase для восстановления базы поддержания секретности isc4.gdb.

Задание прав. Команда GRANT

Создание пользователя само по себе не дает ему никаких прав на доступ к объектам базы данных.

Права доступа предоставляются командой GRANT. При этом нужно помнить, что пользователь, выдающий команду GRANT, может передать или, если это вам больше нравится делегировать другим пользователям только те права, которыми он обладает сам.

GRANT устанавливает права на объекты базы данных пользователям, ролям или другим объектам базы данных. Когда объект создается, права на него имеет только его создатель и только он может выдавать права другим пользователям или объектам.

Для доступа к таблице или обзору пользователь или объект нуждается в правах на SELECT, INSERT, UPDATE, DELETE или REFERENCES. Все права могут быть даны опцией ALL.

Для вызова процедуры в приложении пользователь должен иметь права на EXECUTE.

Пользователи могут получить разрешение выдавать права другим пользователям передачей прав по списку <userlist>, который задается *оп-*

цией WITH GRANT OPTION. Пользователь может выдавать другим только те права, которыми располагает сам.

Права могут быть даны всем пользователям опцией PUBLIC на месте списка имен пользователей. Указание опции PUBLIC распространяется только на пользователей, а не на объекты базы данных.

Перечень прав приведен в табл. 8.5.

Таблица 8.5. *Перечень прав*

Право	*Позволяетпользователям*
ALL	SELECT, DELETE, INSERT, UPDATE, EXECUTE. и REFERENCES (последнее только для версий позже 4)
SELECT	Дает право выбирать строки из таблицы или представления
DELETE	Дает право удалять строки из таблицы или представления
INSERT	Дает право добавлять строки в таблицу или представления
UPDATE	Дает право изменять строки в таблице или представлении. Может быть ограничено только определенным набором столбцов
REFERENCES	Дает право ссылаться при работе с внешним ключом на специфицированные столбцы; как минимум это должны быть все столбцы первичного ключа
EXECUTE	Выполнять хранимую процедуру

Права могут быть ликвидированы пользователем, выдавшим их, через команду REVOKE. Если права были выданы с помощью ALL, то и ликвидированы они могут быть только в режиме ALL, если права были выданы с помощью PUBLIC, то и ликвидированы они могут быть только в режиме PUBLIC.

Синтаксис:

```
GRANT {ALL [PRIVILEGES] / LIST_<privilege>} ON [TABLE]
{tablename / viewname}
TO {<object> / LIST_<user> / GROUP UNIX_group}
/EXECUTE ON PROCEDURE procname TO
  {LIST_<object> LIST_<user> [WITH GRANT OPTION./}
/LIST_rolename TO {PUBLIC
  / LIST_<grants> [WITH ADMIN OPTION] } ;
```

```
<privilege>::= SELECT / DELETE / INSERT
/ UPDATE [(LIST_col)]
/ REFERENCES [(LIST_col)]

<object>::= PROCEDURE procname
/ TRIGGER trigname
/ VIEW viewname
/ PUBLIC

<user>::= [USER] username
/ rolename
/ Unix_user}

<grants>:;= [USER] username
```

Таблица 8.6. *Описание синтаксических элементов* ***команды GRANT***

Аргумент	*Описание*
privilege	Имя предоставленного права. Допустимые значения: SELECT, DELETE, INSERT, UPDATE, REFERENCES
Col	Имя столбца, на который выдаются права.
Tablename	Имя существующей таблицы, на которую распространяются права
Viewname	Имя существующего обзора, на который распространяются права
<object>	Имя существующего объекта базы (процедуры, обзора, триггера), на который распространяются права
username	Имя пользователя, которому передаются права
WITH GRANT OPTION	Передает права на передачу прав пользователям, перечисленным в списке *LIST_*<user>
rolename	Имя существующей роли, созданной командой CREATE ROLE
<grants>	Пользователь, которому передаются права роли. Список пользователей должен быть задан в isc4.gdb (создан, например утилитой IBConsole)
GROUP unix_group	Имя группы в UNIX, заданной в /etc/group

Следующая команда передает права пользователю на SELECT и DELETE. Опция WITH GRANT OPTION дает права на их дальнейшую передачу.

Пример 8.5

```
GRANT SELECT, DELETE ON TBOOK TO MISHA WITH GRANT OPTION;
```

А эта команда дает право на выполнение процедуры другой процедуре и пользователю.

Пример 8.6

```
GRANT EXECUTE ON PROCEDURE PAUTHOR
 TO PROCEDURE PBOOKAUTHOR, MISHA;
```

В данном случае передача прав процедуре PBOOKAUTHOR в нашей базе бессмысленна, поскольку она просто не использует процедуру PAUTHOR, но синтаксически она вполне корректна.

Следующая команда по содержанию полностью аналогична примеру 8.5., но ориентирована на использование внедренного SQL.

Пример 8.7

```
EXEC SQL
GRANT SELECT, DELETE ON TBOOK TO MISHA WITH GRANT OPTION;
```

Ликвидация прав. Команда REVOKE

REVOKE ликвидирует права доступа к объектам базы данных. Права - это действия с объектом, которые разрешены пользователю. SQL-права описаны в табл. 8.7.

Таблица 8.7. *Перечень прав*

Право	*Запрещает пользователям*
ALL	SELECT, DELETE, INSERT, UPDATE и EXECUTE
SELECT	Выбирать строки из таблицы или представления
DELETE	Удалять строки из таблицы или представления
INSERT	Вставлять строки в таблицу или представления
UPDATE	Изменять строки в таблице или представлении. Может быть задано для определенного набора столбцов
EXECUTE	Выполнять хранимую процедуру

Право	*Запрещает пользователям*
GRANT OPTION FOR	Делегирование прав другим пользователям

Отметим некоторые ограничения при использовании команды REVOKE. Ликвидировать права может только тот пользователь, кто их выдал. Одному пользователю могут быть переданы одни и те же права на объект базы данных от любого числа разных пользователей. Команда REVOKE влечет за собой лишение выданных ранее именно этим пользователем прав. Права, выданные всем пользователям опцией PUBLIC, могут быть ликвидированы командой REVOKE только с опцией PUBLIC.

Синтаксис:

```
REVOKE [GRANT OPTION FOR]
{ALL [PRIVILEGES] | LIST_<privilege>} ON [TABLE] {tablename | viewname}
        FROM { LIST_<object> / LIST_<user>}
    | EXECUTE ON PROCEDURE procname
        FROM {<object> / LIST_<user>}
    };

<privilege>::= SELECT / DELETE / INSERT
/ UPDATE [(LIST_col)]
/ REFERENCES [(LIST_col)]
<object>::= PROCEDURE procname / TRIGGER trigname / VIEW viewname
    / [USER] username / PUBLIC

<user>::= [USER] username
```

Таблица 8.8. *Описание синтаксических элементов команды REVOKE*

Аргумент	*Описание*
GRANT OPTION FOR	Отменяет у перечисленных пользователей права на делегирование прав на объекты базы данных. Неприменим по отношению к объектам базы данных
col	Имя столбца, на который выдаются права

Аргумент	*Описание*
tablename	Имя существующей таблицы, на которую распространяются права
viewname	Имя существующего представления, на которое распространяются права
<object>	Имя существующего объекта базы, на который распространяются права
<user>	Пользователь, у которого ликвидируются права

Следующая команда ликвидирует у пользователя права на удаление из таблицы (см. пример 8.5, в этом случае права на чтение у него остаются).

Пример 8.8

```
REVOKE DELETE ON TBOOK FROM MISHA;
```

А эта команда отменяет право на выполнение процедуры другой процедуре и пользователю (см. выделение соответствующих прав в примере 8.6)

Пример 8.9

```
REVOKE EXECUTE ON PROCEDURE PAUTHOR
  FROM PROCEDURE PBOOKAUTHOR, MISHA;
```

Создание группы управления правами - роли.

Прежде всего, разберемся с понятием роли. Рассмотрим некоторое множество команд предоставления прав (GRANT) и дадим ему имя. В этом случае вместо перечисления команд GRANT можно, указав имя, сослаться на это множество. Такой подход позволяет одной командой передать пользователю права сразу на несколько объектов. Если пользователей много, то действия по отслеживанию всех их прав становятся достаточно громоздкими и работа с такими множествами существенно ее облегчает. Особенно удобно то, что вместо предоставления прав на какой-либо новый объект или ограничения ранее предоставленных прав каждому из пользователей, можно провести такое изменение в нашем поименованном множестве, которым и является роль.

Процедура работы с ролями включает несколько этапов.

- Создание роли, то есть объявление ее в базе (создание имени и пустого множества).

- Формирование списка прав, связанных с ролью (включение элементов - прав в множество).
- Формирование прав пользователей на основе ролей
- Связывание пользователей с ролями, то есть передача им множества прав, описанных в роли.

КомандыCREATEROLE,DROPROLE

Команда CREATE ROLE реализует первый этап действий при работе с ролями: создает (объявляет) роль в базе данных.

Синтаксис:

```
CREATE ROLE rolename;
```

Таблица 8.9. *Опции команды CREATE ROLE*

Параметр	*Описание*
Rolename	Имя роли; должно быть уникальным среди функциональных имен в базе данных

Пример 8.10

Следующая команда создает роль, называемую "bibrole".

```
CREATE ROLE bibrole;
```

Команда DROP ROLE выполняет действия, обратные CREATE ROLE - удаляет роль из базы данных.

Синтаксис:

```
DROP ROLE rolename;
```

Таблица 8.10. *Опции команды DROP ROLE*

Параметр	*Описание*
Rolename	Имя существующей роли, которая удаляется из базы данных

Роль может быть удалена либо ее создателем, либо пользователем SYSDBA, либо другим пользователем с аналогичными правами.

Пример 8.11

Следующая команда удаляет роль, называемую " bibrole ".

```
DROP ROLE bibrole;
```

Формирование списка прав, связанных с ролью

Ролям, созданным командой CREATE ROLE, могут быть предоставлены права так же, как и пользователям.

Предоставление прав ролям осуществляется командой GRANT. Общий синтаксис команды GRANT уже приводился. Тем не менее, приведем его еще раз применительно к предоставлению прав ролям.

```
GRANT {ALL [PRIVILEGES] / LIST_<privilege>} ON [TABLE]
{tablename / viewname}
TO LIST_<user>
/EXECUTE ON PROCEDURE procname TO
/ LIST_<user> [WITH GRANT OPTION]

<privilege>::= SELECT / DELETE / INSERT
/ UPDATE [(LIST_col)]
/ REFERENCES [ (LIST_col) ]

<user>::= [USER] username
/ rolename
/ Unix_user}
```

Отметим, что одной командой GRANT права могут предоставляться и ролям и пользователям.

Таблица **8.11.** *Описание синтаксических элементов команды GRANT для ролей*

Аргумент	*Описание*
privilege	Имя предоставленного права. Допустимые значения: SELECT, DELETE, INSERT, UPDATE, REFERENCES
Col	Имя столбца, на который выдаются права.
Tablename	Имя существующей таблицы, на которую распространяются права

Аргумент	*Описание*
Viewname	Имя существующего обзора, на который распространяются права
username	Имя пользователя, которому передаются права
WITH GRANT OPTION	Передает права на передачу прав пользователям, перечисленным в списке *LIST_*<user>
rolename	Имя существующей роли, созданной командой CREATE ROLE

Рассмотрим пример предоставления прав (привилегий) роли "bibrole" (см. примеры 8.6 и 8.10) на выполнение процедуры PAUTHOR.

Пример 8.12

```
CREATE ROLE bibrole;
GRANT EXECUTE ON PROCEDURE PAUTHOR
 TO bibrole;
```

Формирование прав пользователей на основе ролей

Сами по себе роли носят вспомогательный характер. С базой данных работают пользователи, а не роли, следовательно, именно пользователям и должны передаваться права на работу с объектами базы данных. Передача прав на объекты базы данных, объявленных для ролей, конечным пользователям осуществляется командой GRANT. Полный синтаксис GRANT описан выше. В части передачи пользователям прав, объявленных для ролей, он выглядит следующим образом.

GRANT *LIST_*rolename **TO**
*{***PUBLIC** / *LIST_*<grants> *[*WITH ADMIN **OPTION***]}*;

<grants>*::= [***USER***]* username

Таблица 8.11. *Описание синтаксических элементов команды GRANT для передачи прав ролей пользователям*

Аргумент	*Описание*
username	Имя пользователя, которому передаются права
rolename	Имя существующей роли, созданной командой CREATE ROLE

Рассмотрим передачу пользователям прав, присвоенных ролям.

Пример 8.13

```
GRANT bibrole TO misha, masha, Ivan_Ivanovitch;
```

Связывание пользователей с ролями

В InterBase с пользователем во время его сеанса работы с базой может быть связана только одна роль. В то же время команд GRANT на передача прав от ролей пользователю может быть несколько.

Такой механизм позволяет динамически связывать набор прав пользователя при его конкретном соединении. Это имеет смысл в тех случаях, когда один и тот же человек выступает в различном качестве, например, сегодня он работает как кассир, а завтра, как приемщик товаров. С этой точки зрения роль можно связать с рабочим местом, а пользователя с человеком.

Таким образом, связь между ролью и пользователем осуществляется не при выдаче команды GRANT, а при соединении пользователя с базой.

Реализация такой связи осуществляется командой CONNECT.

Базовый синтаксис команды CONNECT для нашего случая выглядит следующим образом:

```
CONNECT USER 'username' PASSWORD 'password' ROLE 'role-
name';
```

Таким образом, один и тот же пользователь при входе в систему может получать различные наборы прав.

Пример 8.14

```
CONNECT USER 'MISHA' PASSWORD '12345' ROLE 'bibrole';
```

MISHA получил права на процедуру PATHOR

```
CONNECT USER 'MISHA' PASSWORD '12345';
```

MISHA не получил права на процедуру PATHOR

8.5. Копирование и восстановление базы данных

Регулярное выполнение операций копирования базы предназначено, прежде всего, для обеспечения возможности восстановления данных после сбоев. Учитывая, что база данных InterBase физически организована в виде одного файла (при наличии файлов тени - нескольких файлов), проблем с копированием базы нет. Единственно, о чем следует помнить, так это о том, что при проведении копирования внешними программами все пользователи должны быть отключены от базы.

Простое копирование, несмотря на его быстроту и надежность, хотя и допустимо, но все же не может быть рекомендовано как основной метод. Предпочтительнее выполнять операции копирования, используя средства базы данных. В этом случае одновременно с операцией копирования выполняется и сервисное обслуживание базы данных.

Использование резервной копии InterBase и особенности восстановления утилитой gbak или диспетчером серверов (Server Manager в InterBase 4-5 или IBConsole в InterBase 6) дают ряд преимуществ. При резервном копировании и восстановлении помимо собственно копирования выполняется также ряд дополнительных действий, а именно:

- Выполняется сборка "мусора" (удаляются устаревшие версии записей) и чистка таблицы транзакций от транзакций, завершенных откатом (rollback).
- Балансируются индексы.
- Освобождается пространство, занимаемое удаленными записями, и упаковываются оставшиеся данные.

Это позволяет несколько уменьшить размер базы данных и ускорить работу с данными.

Выполнение функции резервного копирования не требует монопольного режима. Во время выполнения копирования пользователи могут продолжать работу. При этом надо, конечно, помнить, что все данные, внесенные пользователями после начала копирования, в саму копию уже не попадут, но согласованность данных копии гарантируется.

Полученная архивная копия может быть сохранена на любом устройстве. С архивной копии можно восстановить существующую или создать новую базу. При копировании, восстановлении можно выполнить также ряд действий по изменению характеристик базы (размер страницы и ряд других).

В результате копирования средствами InterBase получается платформно-независимый, устойчивый снимок базы.

Благодаря этому данные могут быть переданы в другую операционную систему. Это важно, поскольку различные платформы имеют аппаратно-зависимые форматы файла базы данных и поэтому базы данных не могут быть просто скопированы для переноса на другую платформу. Создание переносимых резервных копий особенно полезно в гетерогенных средах. Необходимо только помнить, что нельзя гарантировать, что копия базы новой версии InterBase пригодна для переноса в старую версию.

Новые версии InterBase могут иметь отличия в физической организации данных. Чтобы модернизировать существующие базы к новой структуре, необходимо:

1. Перед установкой новой версии InterBase скопировать базы данных, использующие старую версию утилит копирования.
2. Установить новую версию сервера InterBase.
3. Как только новая версия установлена, восстановить базы данных с новой версией InterBase. Восстановленные базы данных теперь готовы использовать все новые возможности сервера InterBase.

Функции резервного копирования и восстановления базы данных можно осуществлять несколькими способами:

- Копирование можно выполнить диспетчером серверов (Server Manager) или IBConsole в зависимости от используемой версии. Для этого используются пункты меню Tasks - Backup. Далее выбираются нужные режимы копирования.
- Для восстановления используются пункты меню Tasks - Restore. Далее выбираются нужные режимы копирования.
- Можно также использовать утилиту командной строки **gbak,** хотя при работе в среде Windows это весьма неудобно.

Синтаксис утилиты для копирования имеет следующий вид:

```
gbak [-B] [options] database target
```

Синтаксис утилиты для восстановления имеет следующий вид.

```
gbak {-C|-R} [options] source database
```

То же, но для баз с несколькими файлами

```
gbak {-C|-R}[options] source primary m secondary1[n1 secondary2 [n2]]
```

Таблица 8.11. *Описание синтаксических элементов командной строки GBAK*

Параметр	*Описание*
Database	Имя копируемой или восстанавливаемой базы данных
Source	Имя запоминающего устройства или файла с резервной копией. В UNIX это может также быть stdin, когда gbak читает со стандартного ввода
Target	Имя запоминающего устройства или файла с резервной копией. В UNIX это может также быть stdout, когда gbak пишет в стандартный файл вывода
Primary	Первичный файл при восстановлении с множественными файлами базы данных
M	Длина первичного файла в страницах базы данных; минимальное значение - 200
secondary 1	Первый вторичный файл при восстановлении с множественными файлами базы данных
n1	Длина secondary 1. Если используется только один вторичный файл, то n1 необязателен
secondary2	Следующий вторичный файл, если задано несколько вторичных файлов
n2	Длина secondary2. Длину последнего вторичного файла определять не нужно

Опции gbak при копировании приведены в следующей таблице.

Таблица 8.12. *Описание опций утилиты GBAK при копировании*

Опция	*Описание*
-b[ackup_database]	Копирует базу данных
-co[nvert]	Конвертирует внешние файлы как внутренние таблицы
-e[xpand]	Не создает сжатой копии
-fa[ctor] n	Использует блокирующий коэффициент n для вывода
-g[arbage_collect]	Не "собирает мусор" при копировании

Опция	Описание
-ig[nore]	Игнорирует контрольные суммы при копировании
-l[imbo]	Игнорирует транзакции в «неопределенном состоянии» при копировании
-m[etadata]	Копирует только метаданные
-nt	Создает копию в непереносимом формате
-ol[d_descriptions]	Копирует метаданных в формате старого стиля
-pa[ssword] text	Проверяет текст пароля перед доступом к базе данных
-role name	Соединяется с указанием роли
-t[ransportable]	Создает копию в переносимом формате (значение по умолчанию)
-u[ser] name	Проверяет имя пользователя перед доступом к удаленной базе данных
-v[erbose]	Показывает действия gbak
-y [file | suppress_output]	Подавляет сообщения вывода
-z	Показывает версию gbak и InterBase

Опции gbak при восстановлении приведены в следующей таблице.

Таблица 8.13. *Описание опций утилиты GBAK при восстановлении*

Опция	Описание
-c[reate_database]	Восстанавливает базу данных как новый файл
-bu[ffers]	Размер кэша для восстановленной базы данных
-i[nactive]	Делает индексы неактивными после восстановления
-k[ill]	Не создает никаких ранее определенных теней
-n[o_validity]	Удаляет ограничения целостности из восстановленных метаданных; позволяет восстановить данные, которые иначе вызвали бы нарушение ограничений
-o[ne_at_a_time]	Восстанавливает по одной таблице; полезно для частичного восстановления, если база содержит поврежденные данные

Опция	*Описание*
-p[age_size] n	Устанавливает размер страницы к п байтам (1024, 2048, 4196, или 8192); значение по умолчанию - 1024
-pa[ssword] text	Проверяет текст пароля перед доступом к базе данных
-r[eplace_database]	Восстанавливает базу данных как новый файл или заменяет существующей файл
-username	Проверяет имя пользователя перед доступом к удаленной базе данных; требуется при использовании gbak с клиентской машины
-use_[all_space]	Восстанавливает базу данных со 100 % заполнением на каждой странице данных, вместо значения по умолчанию (коэффициент заполнения 80 %)
-v[erbose]	Показывает действия gbak
-y [file |	Подавляют сообщения вывода
-z	Показывает версию gbak и InterBase

Кроме того, следует отметить, что для выполнения сервисных функций с базами данных можно использовать также утилиты третьих фирм, которые в ряде случаев заметно проще и удобнее в работе (см. гл. 11).

И еще одно замечание. В ряде случаев с помощью таких утилит можно провести операции восстановления или переноса базы еще одним способом: выгрузить базу в виде SQL-скрипта, включающего как команды создания базы и ее объектов, так и содержимого таблиц. При таком способе обеспечивается, как переносимость данных базы, так и возможность ручной корректировки отдельных параметров базы при ее восстановлении.

Глава 9

Транзакции. Механизм транзакций в InterBase

9.1. Понятие транзакции. Назначение транзакций

Транзакции и поддержание логической целостности данных

Прежде всего, определимся с понятием транзакции (transaction). При внесении изменений в базу данных возникает ряд проблем, даже если с базой работает только один пользователь. Данные одного документа могут в базе храниться в различных таблицах, кроме того, они могут использовать другие данные, логически связанные с ними. Если обработка документа будет по тем или иным причинам прервана, то это может привести не только к неполноте данных, но и к нарушению их логической целостности. Например, агрегированные данные могут разойтись с исходными данными, на основе которых они были получены. При обработке множества строк таблицы возможна ситуация, когда часть строк обработана, а другая нет, например, изменение размера пенсий для определенной группы. Простой повтор таких операций невозможен, поскольку неизвестно, в какие именно строки были внесены изменения, а какие изменены не были. Другими словами, неполный ввод (модификация или Удаление) логически связанных групп данных чреват большими трудностями по восстановлению целостности и непротиворечивости информации. Для решения этой проблемы и предусматривается использования механизмов управления транзакциями.

Транзакция - это группа операций с базой данных, выполняемых как единое целое. Запись данных в базу производится только при успешном выполнении всех операций группы. Если хотя бы одна из операций группы завершается неуспешно, то база данных возвращается к тому состоянию, в котором она бьша до выполнения первой операции группы (производится откат всех изменений). Таким образом, после устранения причины неудачного выполнения групповой операции с базой ее можно просто повторить. То есть механизм транзакций позволяет обеспечить логическую целостность данных в базе. Другими словами, транзакции - это логические единицы работы, после выполнения которых база данных остается *в целостном состоянии*. Транзакции также являются *единицами восстановления* данных. После сбоев - восстанавливаясь, система ликвидирует следы транзакций, не успевших успешно завершиться в результате программного или аппаратного сбоя. Эти два свойства транзакций определяют атомарность (неделимость) транзакции.

Логическая целостность базы включает два уровня: формальную целостность и семантическую (смысловую) целостность.

Под формальной логической целостностью понимается соблюдение явно описанных в базе ограничений логической целостности, таких как:

- ограничения первичных ключей (PRIMARY KEY);
- ограничения уникальных ключей (UNIQUE KEY);
- ограничения внешних ключей (FOREIGN KEY);
- ограничения, задаваемые конструкциями CHECK;
- ограничения, задаваемые используемыми триггерами.

Обеспечение целостности на этом уровне осуществляется стандартными средствами базы. Пользователю просто не удастся ввести данные, нарушающие эти ограничения.

Под семантической логической целостностью будем понимать соблюдение требования полноты внесения группы логически связанных изменений, обеспечивающих согласованность данных в базе. Выполнение именно этого требования и реализуется с помощью механизма транзакций.

Проблемы доступа к данным в многопользовательских системах

Механизм транзакций в однопользовательских системах покоится на трех китах: *атомарность* - транзакция выполняется как единое целое, *согласованность* - в результате выполнения транзакции база данных переходит из одного согласованного состояния в другое, *долговечность* - результаты транзакции сохраняются в базе данных.

Если система работает только с одним пользователем, то использование транзакций гарантирует логическую целостность данных, проблем же совместного доступа здесь просто нет. Если же с системой работает несколько пользователей, то картина существенно меняется.

При наличии нескольких пользователей в системе может одновременно существовать несколько транзакций, а раз так, то они могут обращаться к одним и тем же данным. Транзакции, пересекающиеся по времени и обращающиеся к одним и тем же данным называются *конкурирующими*.

Следовательно, необходимо позаботиться о том, чтобы одна транзакция не могла менять данные, уже измененные другой транзакцией, пока та не завершилась. Поэтому перечисленных трех *китов* в многопользовательской системе недостаточно. К ним необходимо добавить еще *черепаху* или, если вам так больше нравится, четвертого кита - *изолированность*. Изолированность транзакций предполагает, что каждая из транзакций должна выполняться так, как если бы она была единственной. Полная изоляция транзакций может быть легко обеспечена запретом запуска следующей транзакции, пока не завершена предыдущая. Такое решение, однако, крайне неэффективно, поэтому в реальных системах требования к изоляции транзакций снижаются.

Рассмотрим сначала возможные конфликты доступа к данным:

- ***W–W (Запись–Запись).*** Первая транзакция изменила объект и не закончилась. Вторая транзакция пытается изменить этот объект. Результат - потеря обновления. При этом может быть нарушена согласованность хранимых данных. Допускать подобное обновление явно нельзя. Данные, измененные какой-либо транзакцией, должны быть защищены от любых изменений до ее завершения.
- ***R–W (Чтение-Запись).*** Первая транзакция прочитала объект и не закончилась. Вторая транзакция пытается изменить этот объект. Результат - данные, полученные первой транзакцией, не соответствуют хранимым в базе. При повторном чтении они могут оказаться другими. Расчеты, сделанные на их основе, могут оказаться неверными. Сами данные в базе при этом остаются согласованными. В качестве примера рассмотрим следующую ситуацию. Первый пользователь (транзакция 1) считал из базы данных какие-либо данные, например свободные места на авиарейс. Пока он смотрит эти данные, второй пользователь (транзакция 2) может их изменить и зафиксировать изменения (транзакция 2 уже закончилась, а транзакция 1 еще думает). Например, продать билет на этот рейс. В этом случае первый пользователь видит несуществующие данные. Чтобы полностью исключить подобные ситуации, нужно блокировать доступ к данным, если они были запрошены каким-

либо пользователем даже только для чтения, но такая блокировка может сильно тормозить работу системы.

- ***W-R (Запись–Чтение)***. Первая транзакция изменила объект и не закончилась. Вторая транзакция пытается прочитать этот объект. Результат - чтение неподтвержденных данных. В случае отката первой транзакции, это означает, что были прочитаны данные, которых в базе вообще никогда не было.

Конфликты типа R-R (Чтение-Чтение) возникнуть не могут, поскольку данные при чтении не меняются.

В связи с тем, что конфликты между транзакциями неизбежны, а полная их изоляция по соображениям эффективности невозможна, при задании транзакций предлагаются различные компромиссные варианты ограничения доступа к данным, называемые ***уровнями изолированности,*** или ***уровнями изоляции.*** Уровень изоляции определяет, как транзакция взаимодействует с другими, конкурирующими транзакциями.

Отметим, что в случае успешности всех операций транзакции изменения фиксируются ***(commit, committed)*** в базе, в противном случае они отменяются ***(rollback, rolled back)*** и база "откатывается" к состоянию, в котором она была перед началом транзакции.

Стандартом ANSI SQL-92 предусматривается 4 стандартных уровня изолированности транзакций:

- ***Dirty Read*** - "грязное" (или "незафиксированное") чтение. Транзакция может читать не подтвержденные изменения, сделанные в других транзакциях. Например, если транзакции A и B стартовали, и поменяли записи, то они обе видят изменения друг друга. ***InterBase не поддерживает уровень изоляции транзакций Dirty Read.***

- ***Read Committed*** - невоспроизводимое (или неповторяемое) чтение. Транзакция может читать только те изменения, которые были подтверждены другими транзакциями. Например, если транзакции A и B стартовали и поменяли записи, то они не видят изменения друг друга. Транзакция A увидит изменения транзакции B только тогда, когда транзакция B завершится по commit. Повторное чтение данных транзакцией A при этом приведет к тому, что она увидит, вообще говоря, уже другие данные. ***InterBase полностью поддерживает уровень изоляции транзакций Read Committed.***

- ***Repeatable Read*** - воспроизводимое (или повторяемое) чтение. Транзакция видит только те данные, которые существовали на момент ее старта. При повторном чтении будут видны те же самые данные, хотя они могли за это время и измениться. Уровень изоляции ***Repeatable Read*** в чистом виде **не** поддерживается InterBase.

Вместо него ***InterBase поддерживает уровень SNAPSHOT*** (снимок), который хотя и близок к Repeatable Read, но несколько "сильнее". Последнее связано с тем, что InterBase использует "версии" данных, а не их блокировку, действительно гарантируя повторное считывание тех же самых данных.

- *Serialized* - **сериализуемость**. Транзакции выполняются так, как будто никаких других транзакций в этот момент не существует. Или, другими словами, транзакции выполняются так, как будто они выполняются последовательно.

Каждый из уровней изоляции имеет свои достоинства и недостатки. ***Dirty Read*** позволяет оперативно отслеживать все изменения в базе, но это всегда "предварительные результаты" и надо быть готовым к их отмене. Кроме того, при просмотре данных из нескольких таблиц нельзя быть уверенным в согласованности данных (логическая целостность обеспечивается только по завершении транзакции).

Следующий уровень изоляции *Read Committed* гарантирует согласованность всех данных, но не может гарантировать их актуальности, данные могли быть уже изменены, но соответствующая операция еще не была подтверждена *(Committed).* Для получения обновленных данных необходимо выполнять операции повторного чтения. Неповторяемость в общем случае результатов чтения может рассматриваться и как достоинство и как недостаток. Кроме того, при сложной выборке возможно получение и просто неверных данных. Рассмотрим это на примере.

Имеется две транзакции. Первая подсчитывает запас продукции на складах. Поскольку в рамках данной транзакции не производится изменений данных, то она не блокирует работу других транзакций. Вторая реализует перемещение продукции между складами и вносит соответствующие изменения в данные. Рассмотрим действие транзакций во времени.

Таблица 9.1. *Конфликт между конкурирующими транзакциями, неверная сумма*

Этапработыповремени	*Перваятранзакция*	*Втораятранзакция*
1	Старт	-
2	Подсчет запаса на первом складе (300 единиц)	-
3		Старт
4	Подсчет запаса на втором складе (200 единиц)	-

Этапработыповремени	*Перваятранзакция*	*Втораятранзакция*
5	-	Первый этап перемещения с первого склада на пятый. Уменьшение запаса на первом на 50 единиц
6	Подсчет запаса на третьем складе (50 единиц)	-
7	-	Второй этап перемещения с первого склада на пятый. Увеличение запаса на пятом на 50 единиц (180+50=230)
8	Подсчет запаса на четвертом складе (150 единиц)	-
9	-	Завершение транзакции
10	Подсчет запаса на пятом складе (230 единиц)	-
11	Завершение транзакции	-

В итоге первая транзакция насчитала 300 + 200 + 50 + 150 + 230 = = 930. На самом же деле, общий запас составляет 880, причем все операции суммирования проводились с данными, которые были подтверждены.

Уровень изоляции ***Repeatable*** *Read* жестко связан с состоянием базы на момент своего старта. Это обеспечивает возможность почти всегда при повторном чтении видеть те же самые данные. В системах с блокировкой данных это достигается запретом изменения данных, прочитанных транзакцией. В то же время такая блокировка не гарантирует от появления "фантомных данных". Пусть первая транзакция читает из таблицы данные, удовлетворяющие некоторому условию *p.* Другая транзакция после этого уже не может менять данные, удовлетворяющие условию *p,* однако, она может работать с другими данными. В результате она вносит в свои данные изменения, после которых они уже удовлетворят условию ***p***. Транзакция, внесшая изменения, успешно завершается, изменения сохранены в базе. После этого первая транзакция вновь читает *свои* данные и ожидает, что они будут теми же. Но не тут-то было. В результате запро-

са она получит и «добавок» в виде данных, внесенных второй транзакцией. В системах с хранением версий данных, к которым относится InterBase, такого рода фантомов не будет, но и уровень изоляции называется иначе - *SNAPSHOT* (снимок). Данный уровень гарантирует полную согласованность всех полученных данных, но не может гарантировать их "свежести". Часть данных может оказаться устаревшей, все данные были актуальными на момент старта транзакции.

Выбор конкретного уровня изоляции зависит от задач, решаемых транзакцией.

9.2. Реализация механизма транзакций в InterBase

Прежде чем перейти непосредственно к описанию работы с транзакциями в InterBase, необходимо разобраться в организации хранения изменений данных в базе и доступа к измененным данным. Для обеспечения изоляции транзакций и, при необходимости, их корректного отката в InterBase помимо измененных данных хранится также и состояние базы до внесения изменений. Другими словами, в базе хранится одновременно несколько версий данных, причем разные транзакции работают с разными версиями.

Хранение версий данных в InterBase

Прежде всего, отметим, что в отличие от большинства баз данных, InterBase хранит не историю выполнения транзакций, а использует версии строк таблиц, получающихся в результате внесения изменений в базу.

При обновлении (update) строки таблицы InterBase сохраняет сначала старое значение строки, а точнее (для экономии объема хранимых данных), разницу между новой и старой строками. Копия сохраняется, если это возможно, на той же странице, что и основные данные, обеспечивая минимизацию времени доступа к сохраненной версии. Затем InterBase заменяет исходную строку новой версией и создает указатель на старую версию (копию). В главную версию строки записывается идентификатор создавшей ее транзакции. Вообще, любая строка таблицы содержит идентификатор создавшей ее транзакции: при создании новой строки в нее помимо самих данных помещается идентификатор создавшей ее транзакции.

При удалении строка физически не удаляется, а лишь помечается как удаленная, с указанием идентификатора удаляющей ее транзакции (старое значение с указанием соответствующей транзакции также сохраняется). В случае необходимости отката транзакции достаточно заменить текущее значение строки таблицы на непосредственно предшествующую версию.

Идентификаторы транзакциям присваиваются таким образом, что транзакция, которая стартовала позже, будет иметь и больший идентификатор.

Чтобы правильно работать с версиями, необходимо располагать информацией о текущем состоянии транзакций. InterBase хранит сведения о текущем состоянии транзакций на специальных страницах базы данных - Transaction Inventory Page (TIP). Транзакция вне зависимости от ее уровня изоляции может находиться в одном из четырех состояний: *active, committed, rolled back* или *in limbo.* Текущее состояние транзакции всегда отражается в глобальной TIP. Помимо глобальной TIP существуют также и локальные TIP, используемые транзакциями уровня SNAPSHOT, отражающие состояние TIP на момент их старта.

Подробнее о возможных состояниях мы поговорим ниже, а пока ограничимся констатацией того факта, что знание состояния транзакцией необходимо для управления версиями строк таблиц базы.

Работа с версиями данных в InterBase

В предыдущем разделе мы говорили о механизме создания версий. Каждая транзакция, обновляющая данные, создает новые версии строк таблиц. Если не позаботиться об удалении ненужных версий, то база очень быстро будет состоять только из них и фактически прекратит работу.

В первую очередь выясним, сведения о каких транзакциях нам необходимы, чтобы обеспечить корректную работу с версиями, а также удаление заведомо устаревших версий.

Пусть какая-либо транзакция успешно завершилась *{commit).* Можно ли удалить после этого версии строк таблиц, предшествующих зафиксированному транзакцией состоянию? Ответ будет отрицательным, поскольку SNAPSHOT-транзакции, стартовавшие до завершения данной транзакции, должны использовать именно их. Старые версии, конечно, должны быть удалены, поскольку замусоривают базу, но сделать это можно только тогда, когда будет точно известно, что более они никому не нужны.

Если какая-либо транзакция завершилась аварийно *{rollback),* то можно ли удалить версии строк таблиц, созданных данной транзакцией? Да, можно. Пока она была активна, никто не мог использовать ее версии, а теперь они вообще не нужны. Нужно ли выполнять откат немедленно? Наверное, нет. Процедура эта довольно громоздка, а мусор в базе все равно сохраняется, и его сборка должна проводиться, так что целесообразнее поручить это процедуре сборки мусора, чем строить еще одну процедуру, что на самом деле и реализовано в InterBase.

В какой же момент можно установить, какие версии потеряли актуальность. Рассмотрим эту проблему подробнее.

Стартует транзакция с уровнем изоляции SNAPSHOT, и пусть ее номер есть N_1. При старте транзакция должна зафиксировать состояние базы данных, определив какие версии данных на момент ее старта являются окончательными, а какие еще могут меняться. Изменения могут вносить только активные транзакции. Пусть транзакция N_{O1} - старейшая из активных транзакций на момент старта данной.

Транзакция читает данные. Рассмотрим ее действия при чтении в зависимости от того, какой транзакцией была создана считанная строка (запись), успешно ли завершилась создавшая ее транзакция и какие версии имеет считанная запись.

- Запись создана транзакцией, завершенной по *rollback* (это проверяется по TIP). Удаляем строку и записываем на ее место предшествующую версию. Здесь мы выполняем часть действий по откату транзакции, причем никакой поиск нам не нужен, а значит, действие может быть выполнено очень быстро. После этого вновь повторяем анализ состояния версий строки.
- Если запись имеет версии, находим старейшую версию, иначе переходим к собственно чтению. Смотрим теперь, какой транзакцией создана эта версия и можно ли эту версию удалить. Удаление не должно нарушить работу ни **одной из активных,** то есть не завершенных по *commit* или *rollback,* транзакций. Следовательно, тот факт, что данная версия не нужна текущей транзакции, еще не означает, что эту версию можно удалить. Проверять все транзакции, конечно, не нужно. Достаточно убедиться, что версия не нужна самой старой из активных в данный момент транзакций. Пусть номер старой из активных на данный момент транзакций N_2, а самая старая активная транзакция на момент старта есть N_2 - N_{o2}. Тогда, если версия создана транзакцией с номером $M<N_{o2}$, ее можно удалить, в противном случае - нельзя. После удаления, если это возможно, всех устаревших версий можно перейти, наконец, к собственно чтению.
- Для транзакций с уровнем изоляции SNAPSHOT можно читать только записи, существовавшие на момент старта транзакции. Итак, проверяем, какой транзакцией создана текущая запись. Если номер создавшей ее транзакции $M<N_{O1}$, то запись может быть прочитана. В противном случае просматривается предшествующая версия, и это действие осуществляется до тех пор, пока не будет найдена версия, удовлетворяющая указанному условию. Отметим, что возможна ситуация, когда такой версии просто нет. Это означает, что данная запись была создана уже после старта транзакции и для нее просто не существует. В этом случае чтение не

производится и осуществляется переход к следующей записи, если такая есть.

Теперь несколько слов о терминологии. Следует признать, что сложившаяся терминология довольно неудачна, особенно в переводе на русский, тем не менее, будем ее придерживаться с одним небольшим исключением, которое оговорим чуть дальше.

Активная транзакция. Транзакция, которая стартовала и не была завершена (по commit - фиксация или rollback - отмена).

Заинтересованная транзакция. Транзакция, которая не была завершена фиксацией (по commit).

Актуальная транзакция. Транзакция, для которой версии записей, созданных ею, должны использоваться активными транзакциями. Данный термин не является стандартным, но само понятие достаточно важно для того, чтобы ввести его как самостоятельное.

Множество активных транзакций, таким образом, является подмножеством заинтересованных транзакций и подмножеством актуальных транзакций.

Чтобы следить за состоянием версий в базе данных, сведения о текущем состоянии транзакций в TIP должны содержать объединение множества заинтересованных транзакций и множества актуальных транзакций. Особую роль в этих множествах играют граничные транзакции.

Старейшая активная транзакция. Это такая активная транзакция, которая стартовала раньше всех других, или, что то же самое, активная транзакция с наименьшим номером.

Старейшая заинтересованная транзакция. Это такая заинтересованная транзакция, которая стартовала раньше всех других, или, что то же самое, заинтересованная транзакция с наименьшим номером.

Старейшая актуальная транзакция. Старейшей актуальной транзакцией будем называть ту актуальную транзакцию, которая стартовала раньше всех других, или, что то же самое, актуальную транзакцию с наименьшим номером.

Замечание. В документации по InterBase *старейшую актуальную транзакцию* называют *старейшей активной транзакцией.* На мой взгляд, ни к чему, кроме недоразумений, это привести не может, поэтому здесь будем использовать термин актуальная транзакция.

Старейшая заинтересованная транзакция - это либо старейшая активная транзакция, либо старейшая из отмененных транзакций (rollback).

При старте каждой транзакции N фиксируется транзакция, которая является на момент ее старта старейшей активной f(N), а для транзакций с уровнем изоляции SNAPSHOT - полный список активных транзакций на момент ее старта. Это необходимо для того, чтобы определить, версиями каких транзакций можно пользоваться. При этом версии более старых транзакций для данной заведомо не нужны. Пусть $N_o(t)$ - старейшая ак-

тивная транзакция на момент времени t. Тогда можно утверждать, что версии транзакций, более старых, чем $N_a(t)=f(N_o(t))$ не нужны ни для одной транзакции, а значит, могут быть удалены. Транзакция $N_a(t)$ и будет старейшей актуальной. Можно сказать, что старейшая актуальная транзакция на заданный момент времени - это транзакция, которая была старейшей активной на момент старта старейшей активной на данный момент времени транзакции.

Теперь рассмотрим порядок обновления данных.

Прежде всего, как уже отмечалось выше, нельзя обновлять версию незавершенной транзакции (конфликт W-W, или Запись-Запись). Но это не все. Транзакция не может обновить версию другой, даже завершенной транзакции, если она стартовала раньше ее, поскольку она при обновлении базируется на более старых данных. Это, естественно, относится только к транзакциям уровня SNAPSHOT. Транзакции уровня READ COMMITED видят самые свежие версии незавершенных транзакций.

Проиллюстрируем все вышесказанное на примере.[1]

Таблица 9.2. *Пример работы с версиями данных в InterBase*

Событие	*Старейшая*			*Комментарий*
	активная	*заинтересованная*	*актуальная*	
Транзакция N стартует	N	N	n=f(N)=N	f(N)=N
Транзакция N создает запись	N	N	N	Это простая вставка. Сервер находит страницу с достаточным местом для хранения записи и ее заголовка и помещает на ней запись, маркируя ее идентификатором транзакции N (TID)
Транзакция N+1 стартует	N	N	N	f(N+1)=N
Транзакция N завершается commit	N+1	N+1	n=f(N+1)=N	

[1] Приведенный пример базируется на материале опубликованной в интернете статьи Ann Harrison «**Как работает версионность данных**» с комментариями Д. Кузьменко. См. ib.demo.ru.

Событие	Старейшая			Комментарий
	активная	заинтересованная	актуальная	
Транзакция N+2 стартует	N+1	N+1	N	f(N+2)=N+1
Транзакция N+3 стартует	N+1	N+1	N	f(N+3)=N+1
Транзакция N+2 создает запись	N+1	N+1	N	Это простая вставка. Создается запись и маркируется идентификатором транзакции N+2 (TID)
Транзакция N+3 модифицирует запись, созданную транзакцией N	N+1	N+1	N	Транзакция N+3 создает копию существующей записи, то есть вычисляет разницу для воспроизведения версии, созданной транзакцией N, помечает обратную версию записи номером транзакции N и записывает ее на свободное место. Затем заменяет исходную запись, сохраненную транзакцией N (и маркированную идентификатором N), новой версией записи, маркированной идентификатором транзакции N+3. Новая запись содержит указатель на обратную версию предыдущей версии записи
Транзакция N+4 стартует	N+1	N+1	N	f(N+4)=N+1
Транзакция N+2 завершается по rollback	N+1	N+1	N	Транзакция N+2 помечается в TIP как отмененная
Транзакция N+5 стартует	N+1	N+1	N	f(N+5)=N+1
Транзакция N+3 завершается commit	N+1	N+1	N	

Событие	Старейшая			Комментарий
	активная	заинтересованная	актуальная	
Транзакция N+5 читает запись, созданную транзакцией N+2	**N+1**	**N+1**	N	Транзакция N+2 помечена в TIP как отмененная. Транзакция N+5 удаляет версию, созданную транзакцией N+2. Обратной записи она не имеет и транзакция N+5 ничего не считывает
Транзакция N+6 стартует	**N+1**	**N+1**	N	f(N+6)=N+1
Транзакция N+1 перечитывает данные таблицы	**N+1**	**N+1**	N	Когда транзакция стартует, то она получает снимок состояний заинтересованных и активных транзакций, который позволяет определить доступные для чтения версии записей. На уровне изоляции snapshot можно видеть только те версии записей, которые были сохранены (committed) до ее старта. Поэтому транзакция N+1 не видит версию транзакции N. Разумеется, сервер не передает клиенту "лишние" версии. Он читает главную версию - на текущий момент это версия транзакции N+3. Затем сервер проверяет состояние этой транзакции в локальном **TIP** (локальном для транзакций repeatable read и глобальном для read committed) и не обнаруживает там N+3-ей транзакции. Проверяет запись и находит там указатель на обратную версию. Читает обратную версию и обнаруживает, что она была создана транзакцией N, которая есть в локальном TIP, но не в состоянии committed, а в active. Поскольку далее обратных указателей нет, то никаких записей клиенту не возвращается

Событие	Старейшая			Комментарий
	активная	заинтересованная	актуальная	
Транзакция N+4 читает таблицу				Когда стартовала транзакция N+4, то транзакция N уже была в состоянии committed, поэтому транзакция N+4 видит версию записи, созданную транзакцией N. Сервер производит те же действия, что и в предыдущем случае, но обнаруживает, что версии записей транзакции N могут быть считаны транзакцией N+4, и поэтому возвращает соответствующие записи
Транзакция N+6 читает таблицу	N+1	N+1	N	Когда стартовала транзакция N+6, транзакции N и N+3 были завершены по commit. Сервер находит и возвращает версию записи транзакции N+3 для транзакции N+6. Он также обнаруживает, что у этой версии есть указатель на обратную версию, и сверяется с глобальной таблицей активных транзакций, чтобы определить, не устарела ли эта версия записи. Поскольку старейшая актуальная транзакция - N, то версия записи транзакции N не является устаревшей, и транзакция N+6 (сервер) оставляет эту версию на своем месте
Транзакция N+5 пытается обновить запись, созданную транзакцией N	N+1	N+1	N	Транзакция N+5 стартовала после завершения (commit) транзакции N, но после старта транзакции N+3. Сервер обнаруживает, что транзакция N+5 пытается обновить старую версию записи, в то время как уже существует новая версия записи. Транзакциям не разрешается перезаписывать изменения конкурирующих транзакций. Поэтому сервер проверяет состояние транзакции N+3. Если она на момент старта транзакции N+5 была активна, то транзакция N+5 должна либо подо-

Событие	Старейшая			Комментарий
	активная	заинтересованная	актуальная	
				ждать, либо немедленно получить сообщение об ошибке (зависит от параметра wait транзакции N+5). Если был указан параметр wait, то транзакция N+5 будет ждать завершения транзакции N+3 по commit или rollback. Если транзакция N+3 отменена, то транзакция N+5 успешно завершит обновление и сможет продолжать дальше. Если транзакция N+3 завершится по commit, то в этот момент транзакция N+5 получит сообщение "update conflict". В нашем случае транзакция N+3 уже завершена по commit, поэтому приложение, стартовавшее транзакцию N+5, немедленно получает сообщение об ошибке. Единственное решение для приложения в данном случае - завершить транзакцию отменой (rollback) и рестартовать в виде транзакции N+7. Если транзакция N+5 не вносила изменений в базу, то лучше ее закончить по commit. В этом случае транзакция не попадет в перечень заинтересованных
Транзакция N+7стартует	N+1	N+1	N	f(N+7)=N+1
Транзакция N+7 пытается обновить запись, созданную транзакцией N	N+1	N+1	N	Поскольку транзакция N+7 стартовала после завершения транзакции N+3, она может читать и обновлять главную версию записи. В конце такого обновления на диске остаются главная версия записи с идентификатором транзакции N+7, обратная версия транзакции N+3 и еще одна обратная версия транзакции N. Обратная версия транзакции N останется на своем месте, поскольку она еще является старейшей актуальной

Событие	Старейшая			Комментарий
	активная	заинтересованная	актуальная	
Транзакция N+1 читает таблицу	N+1	N+1	N	
Транзакция N+4 читает таблицу	N+1	**N+1**	N	**Каждая транзакция видит обратные** версии записей транзакций N+3 **и** N (за исключением транзакции N+1, которая так до сих пор и не может видеть ни одной версии). Каждая транзакция читает только версию, сохраненную на момент ее старта. Все транзакции получают воспроизводимое чтение (repeatable read). Транзакция N+1 не видит ни одной из версий. Транзакция N+4 видит версию транзакции N
Транзакция N+6 читает таблицу	N+1	**N+1**	N	Транзакция N+6 видит версию транзакции N+3
Транзакция N+7 читает таблицу	N+1	N+1	N	Транзакция N+7 видит свою собственную версию (не сохраненную по commit). В это время транзакции N+6 и N+7 видят, что читаемая ими запись имеет указатель на обратную версию, и пытаются определить, не является ли эта обратная версия ненужной. Обе транзакции обнаруживают, что старейшей активной является транзакция N+1, которая не может видеть изменений, произведенных транзакцией N+3. Поэтому никакие обратные версии не являются устаревшими и сборка мусора не производится

Событие	Старейшая			Комментарий
	активная	заинтересованная	актуальная	
Транзакция N+1 завершается commit	N+4	N+2	n=f(N+4)=N+1	Теперь старейшей заинтересованной становится отмененная транзакция N+2
Транзакция N+4 завершается commit	N+5	N+2	n=f(N+5)=N+1	
Транзакция N+7 завершается commit	N+5	N+2	N+1	
Транзакция N+8 стартует	N+5	N+2	N+1	f(N+8)=N+5
Транзакция N+8 читает таблицу	N+5	N+2	N+1	Когда транзакция N+8 (сервер) читает запись, о которой мы говорим уже полчаса, она читает версию, созданную транзакцией N+7. Поскольку у этой версии есть указатель на обратную версию, то транзакция N+8 предполагает выполнение сборки мусора. В соответствии с локальным TIP транзакции N+8 старейшей активной является транзакция N+6. Она может видеть изменения, сделанные транзакцией N+3. Таким образом, версия транзакции N становится никому не нужной и убирается как мусор
Транзакция N+9 стартует	N+5	N+2	N+1	f(N+9)=N+5
Транзакция N+9 модифицирует запись	N+5	N+2	N+1	Все то же самое. Окончательно мы имеем главную версию записи с идентификатором транзакции N+9, первую обратную версию с идентификатором транзакции N+7 и вторую обратную версию с идентификатором транзакции N+3

Событие	Старейшая			Комментарий
	активная	заинтересованная	актуальная	
Транзакция N+9 завершается по rollback	N+5	N+2	N+1	
Транзакция N+10 стартует	N+5	N+2	N+1	f(N+10)=N+1
Транзакция N+10 читает таблицу	**N+5**	N+5	N+1	В копии TIP транзакции N+10 записано, что транзакция N+9 завершена по rollback. Когда она читает запись транзакции N+9, то видит указатель на обратную версию, и применяет обратную версию к главной записи. Этот процесс восстанавливает запись транзакции N+7 в оригинальном виде. Транзакция N+10 помещает эту версию записи на место главной записи (там, где была версия транзакции N+9), а указатель обратной версии теперь показывает на обратную версию транзакции N+3. Обратная версия транзакции N+3 опять не сдвинулась с места, поскольку старейшая актуальная по-прежнему N+1

Сборка мусора и чистка

Сделаем несколько замечаний о работе InterBase с версиями данных:

- При всяком изменении данных создается обратная копия (со сжатием за счет записи только измененной части) записей - версия записи.
- Ненужные, то есть устаревшие, данные, а также данные отвергнутых транзакций удаляются при чтении данных. Подобный процесс называется кооперативным восстановлением. Отдельной процедуры сборки мусора просто не существует.
- Не существует и автоматического алгоритма, гарантирующего, что все устаревшие данные удалены. Это приводит к тому, что нельзя удалить из TIP ни одну отмененную транзакцию (все отме-

ненные транзакции остаются заинтересованными). Таким образом, если не принимать специальных мер, размер TIP будет неограниченно возрастать.

Из сказанного следует, что помимо кооперативной процедуры сборки мусора, необходима также процедура, обеспечивающая периодическую чистку TIP от ненужных сведений об отмененных транзакциях.

Таким образом, для поддержания разумного размера TIP необходимо периодически проводить процедуру чистки. Частая чистка будет замедлять работу с базой. Редкая чистка будет приводить к росту TIP, увеличению размера базы и, в конечном итоге, также к замедлению работы. По умолчанию, чистка производится, когда разница между старейшей заинтересованной и старейшей актуальной достигнет 20000. Данная величина (Sweep Interval) может быть, при желании, изменена. Удобнее всего это можно сделать, используя утилиту **InterBase Server Manager.**

Что происходит при выполнении чистки?

Чистка удаляет все изменения, сделанные отмененными *(rollbacked)* транзакциями, после чего меняет их состояние на подтвержденное (*commited*). В самом деле, после того как удалены все изменения, произведенные отмененной транзакцией, уже не имеет значения, каким образом она была завершена, и можно считать, что она была завершена по *commit.* В то же время подтвержденная транзакция уже не является заинтересованной, следовательно, размер используемого TIP может быть существенно сокращен.

Итак, чистка - это не сборка мусора. Все, что делает чистка, это перемещение старейшей заинтересованной транзакции "вверх", и уменьшение размера маски транзакций. В то же время чистка собирает мусор, как и любая другая транзакция.

Режимы работы транзакций

Работа транзакций может существенно отличаться в зависимости от того, в каких режимах она работает.

В InterBase выделяется по своему назначению четыре группы режимов:

- Режим доступа к данным *(READ WRITE* или *READ ONLY).*
- Режим обработки конфликтов доступа *(WAIT* или *NO WAIT).*
- Уровень изоляции *(READ COMMITED RECORD_VERSION, READ COMMITED NO RECORD_VERSION, SNAPSHOT, SNAPSHOT TABLESTABILITY).*
- **Режим блокировки** *(SHARED READ, SHARED WRITE, PROTECTED READ, PROTECTED WRITE).*

Режим доступа к данным **READ WRITE / READ ONLY.** Операторы внутри транзакции могут или не могут модифицировать данные. Если не указано явно, то по умолчанию принимается *READ WRITE,* то есть, разрешено и чтение и запись. Если транзакция стартует с режимом *READ ONLY,* то любые операции изменения данных будут вызывать сообщение об ошибке. Если транзакция не изменяет данные, то никаких дополнительных затрат во время ее прохождения при старте с режимом *READ WRITE* не возникает, поскольку она не создает новых версий записей. Единственное, о чем необходимо помнить, так это о том, что не следует завершать транзакции не меняющие данные по *rollback,* чтобы не оставлять их по завершении как заинтересованные.

Режим обработки конфликтов доступа **WAIT / NO WAIT.** Если транзакция стартует в режиме *WAIT* (по умолчанию) и при выполнении операции обнаруживается конфликт, то операция "замораживается" до разрешения конфликта. Режим *WAIT* имеет смысл только, если уровень изоляции позволяет изменять ранее заблокированные строки, то есть является уровнем *READ COMMITTED.* В режиме *SNAPSHOT,* установленном по умолчанию, ожидание бесполезно. Кроме того, приложение, выполняющее запрос в таком режиме, «подвисает» на время ожидания. Рекомендуется использовать *NO WAIT* с обработкой кода ошибки. В режиме *NO WAIT* сообщение о конфликте выдается приложению немедленно (возникает ошибка), а операция, которая привела к конфликту, отменяется. В случае взаимоблокировки двух wait-транзакций сервер автоматически обнаруживает эту ситуацию и разблокирует одну из транзакций, будто она стартовала в режиме nowait, через интервал времени, определенный параметром *DEADLOCK_TIMEOUT* в *IBCONFIG,* который по умолчанию равен 10 секундам.

Уровень изоляции **SNAPSHOT** эквивалентен Repeatable Read. На самом деле этот уровень изоляции ближе к "изолированности образа", так как не допускает фантомов, то есть фактически соответствует уровню *SERIALIZABLE.* Все операции в транзакции с данным уровнем изоляции видят только те данные, которые существовали на момент старта этой транзакции (даже если они впоследствии были изменены или удалены другими транзакциями). Уровень изоляции *SNAPSHOT* принимается по умолчанию.

Уровень изоляции **SNAPSHOT TABLE STABILITY** аналогичен уровню изоляции *SNAPSHOT,* но фактически блокирует всю таблицу на запись. Другие транзакции могут только читать. Транзакция *SNAPSHOT TABLE STABILITY* перед чтением из таблицы пытается поставить на нее блокировку *protected-read*; перед записью в таблицу пытается поставить на нее блокировку *protected-write*.

Основное отличие транзакций этого типа от транзакций *SNAPSHOT* в том, что они ставят на таблицы блокировки *protected* (защищенные),

а не *shared* (разделяемые). Виды блокировок можно указать и явно, используя соответствующий режим.

При блокировке таблицы можно быть уверенным, что, если блокировка прошла, пройдут и все обновления (конкурентов быть не может). Конкурирующие транзакции будут либо сразу же отвергнуты, либо будут ждать окончания транзакции *SNAPSHOT TABLE STABILITY.*

Уровень изоляции **READ COMMITED RECORD_VERSION** разрешает чтение только подтвержденных *(committed)* данных. Именно этот режим принимается в BDE по умолчанию. При этом производится чтение последней подтвержденной версии записи, даже если существует более поздняя, но не подтвержденная запись.

Уровень изоляции **READ COMMITED NO RECORD_VERSION** запрещает чтение измененных, но не подтвержденных (committed) записей. При чтении таких записей возникает ошибка *"deadlock".* В режиме *WAIT* транзакция будет ожидать завершения или отмены транзакции, изменившей данные. Чтение каких-либо "старых" версий данных не производится. Для *READ COMMITTED* режим *NO RECORDJVERSION* принимается по умолчанию.

Режимы блокировки дают возможность транзакциям гарантировать себе определенные уровни доступа к таблицам за счет других параллельно выполняющихся транзакций. Блокировки включаются при старте транзакции, а не тогда, когда командам манипулирования данными фактически потребуется соответствующая блокировка. Необходимость в блокировке может возникнуть только в среде, где одновременно несколько транзакций используют одни и те же данные. Блокировки используются для решения следующих трех основных задач:

1. Предотвращения возможных тупиков и конфликтов модификаций, которые могут возникнуть, если блокировки устанавливаются только, когда это становится фактически необходимым, такое поведение задается по умолчанию.
2. Обеспечения *зависимой блокировки,* то есть блокировки таблиц, которая может потребоваться для выполнения действий в триггерах и ограничениях целостности. До тех пор пока явная блокировка не требуется, только такая блокировка может гарантировать, что конфликты модификаций не произойдут из-за косвенных конфликтов таблиц.
3. Изменения уровня распределенного доступа для одной или нескольких таблиц в *READ WRITE.* Например, транзакция *SNAPSHOT READ WRITE* может нуждаться в исключительных правах модификации для отдельной таблицы. Фиксация этих прав и осуществляется заданием режимов блокировки.

Режим блокировки **PROTECTED READ** запрещает другим транзакциям модифицирование строк таблиц. Чтение разрешено любым транзакциям.

Режим блокировки **PROTECTED WRITE** запрещает другим транзакциям модифицирование строк таблиц. Чтение разрешено транзакциям *SNAPSHOT* и *READ COMMITTED,* обновление - только данной.

Режим блокировки **SHARED READ** разрешает чтение любым тран-; закциям. Любые *READ WRITE* транзакции могут обновлять таблицы. Это наиболее либеральный режим блокировки.

Режим блокировки **SHARED WRITE** разрешает любым *SNAPSHOT* и *READ COMMITTED READ WRITE* транзакциям обновлять таблицы. Остальные *SNAPSHOT* и *READ COMMITTED* транзакции могут читать данные.

Совместимость различных видов блокировок показана в таблице.

Таблица 9.3. *Совместимость различных видов блокировок*

	shared_read	*shared_write*	*protected_read*	*protected_write*
shared_read	Да	Да	да	да
shared_write	Да	Да	Нет	Нет
protected_read	Да	Нет	Да	Нет
protected_write	Да	Нет	Нет	Нет

Транзакции, работающие с несколькими базами

НЕОПРЕДЕЛЕННЫЕ(LIMBO)ТРАНЗАКЦИИ

При завершении транзакции, работающей с несколькими базами данных, InterBase автоматически выполняет двухфазный commit. Двухфазный commit гарантирует, что изменения будут внесены либо во все вовлеченные базы данных, либо не будут внесены ни в одну из них, исключая возможность частичной модификации.

Примечание. Borland Database Engine (BDE) не обеспечивает двухфазный commit (транзакций, работающих с несколькими базами InterBase), поэтому приложения, использующие BDE, никогда не создают limbo-транзакции.

ОПРЕДЕЛЕНИЕLIMBOТРАНЗАКЦИИ

На первой стадии двухфазного завершения *(commit),* InterBase готовит каждую базу данных для завершения, записывая изменения от каждой

подтранзакции в базы данных. Подтранзакция - часть транзакции с несколькими базами данных, которая работает только с одной базой.

На второй стадии InterBase отмечает каждую подтранзакцию как завершенную (*commited*) в том порядке, в каком они были подготовлены.

Если при двухфазном завершении возникает сбой на второй стадии, некоторые подтранзакции оказываются завершены, а другие - нет. Двухфазное завершение может окончиться аварийно при разрыве связи в сети или дисковом отказе, делающих одну или несколько баз недоступными. Отказ в двухфазных транзакциях оставляет ее в неопределенном *(limbo)* состоянии, когда сервер не знает, следует ли завершить *{commit)* или отменить *(rollback)* изменения.

Возможно, что некоторые записи в базе данных являются недоступными, поскольку изменившая их транзакция находится в состоянии неопределенности. Чтобы устранить это, следует вернуть транзакцию, используя диспетчер серверов *(Server Manager)*. Возврат транзакции в данном случае означает выполнение фиксации записи *(commit)* или отката *(rollback)*.

9.3. Синтаксис установки параметров транзакции

Описание транзакции, специфицирующее режимы ее работы, включая порядок обработки конфликтов, уровень изоляции, порядок блокировки таблиц, осуществляется командой SET TRANSACTION. Команда SET TRANSACTION описывает и стартует транзакцию.

```
SET TRANSACTION [NAME transaction]
[READ WRITE | READ ONLY]
[WAIT | NO WAIT]
[[ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY]
| READ COMMITTED [[NO] RECORD_VERSION]}]
[RESERVING <reserving_clause>
| USINGdbhandle [, dbhandle ...] ] ;
<reserving_clause> = table [, table ...]
[FOR [SHARED | PROTECTED] {READ | WRITE}] [, <reserv-
ing_clause>]
```

Таблица 9.4. *Описание синтаксических конструкций команды SET TRANSACTION*

Резким	***Описание***
NAME transaction	Задает имя транзакции. < transaction > - предварительно объявленная и инициализированная переменная базового языка. Только SQL
READ WRITE	Определяет, что транзакция может и читать и записывать данные. (Значение, принимаемое по умолчанию)
READ ONLY	Определяет, что транзакция может только читать таблицы
WAIT [Default]	Определяет, что транзакция при конфликте с другой транзакция ждет ее завершения для получения доступа к данным
NO WAIT	Определяет, что транзакция немедленно возвращает сообщение об ошибке, если возникает конфликт блокировок

SET TRANSACTION стартует транзакцию, задает тип доступа к данным в базе данных, поведение при конфликте блокировок, порядок взаимодействия с другими параллельными транзакциями (уровень изоляции), обращающимися к тем же самым данным. Команда может также резервировать блокировки для таблиц. В качестве альтернативы резервированию таблиц приложения, работающие с несколькими базами данных, могут ограничить доступ транзакции к подмножеству подключенных баз данных.

Приложения, подготавливаемые препроцессором gpre с режимом "manual" должны явно запускать каждую транзакцию командой SET TRANSACTION.

SET TRANSACTION воздействует на заданную по умолчанию транзакцию, если ее имя явно не задано в необязательном предложении NAME. Задание поименованных транзакций обеспечивает возможность одновременного использования одним приложением нескольких транзакций. Все имена транзакций должны быть объявлены как переменные базового языка во время компиляции. В DSQL это ограничение предотвращает динамическую спецификацию имен транзакций.

По умолчанию транзакция имеет доступ к базе как на чтение, так и на запись (READ WRITE). Если транзакция должна только читать данные, доступ к базе следует определять как READ ONLY.

Когда одновременно работающие транзакции пытаются модифицировать те же самые данные в таблицах, только первая модификация завершается успешно. Никакая другая транзакция не может модифициро-

вать или удалять эти данные, пока транзакция, выполнившая модификацию, не зафиксирована (commit) или не отменена (rollback). По умолчанию, транзакция ждет окончания конкурирующей транзакции, а затем делает попытку выполнения собственных действий. Чтобы транзакция немедленно прерывалась и сообщала о конфликте блокировки, следует в ее описании указывать параметр NO WAIT.

Уровень изоляции определяет, как транзакция взаимодействует с другими одновременно выполняющимися транзакциями, обращающимися к тем же самым таблицам. Заданный по умолчанию уровень изоляции – SNAPSHOT (снимок). Этот уровень обеспечивает многократно воспроизводимое чтение базы данных по состоянию на момент старта транзакции. Изменения, внесенные другими одновременно выполняющимися транзакциями, не видны.

Уровень изоляции SNAPSHOT TABLE STABILITY обеспечивает воспроизводимое чтение базы данных, гарантируя, что другие транзакции не могут записывать в таблицы, хотя и могут читать из них.

Уровень изоляции READ COMMITTED дает возможность транзакции видеть последние изменения, внесенные другими транзакциями сразу же по их завершении. Транзакция может также модифицировать строки, пока не происходят конфликты модификаций. Неподтвержденные изменения, то есть те, по которым не был выполнен commit или rollback, доступны только во внесшей их транзакции. Для других транзакций они остаются невидимыми до их завершения. Порядок доступа к данным на уровне изоляции READ COMMITTED дополнительно регулируется двумя необязательными параметрами.

NO RECORD_VERSION (принимается по умолчанию) читает только самую последнюю версию строки, доступ к версиям записи запрещен. Если задана опция WAIT, то транзакция ждет, пока самая последняя версия строки не будет подтверждена (commit) или отменена (rollback) модифицировавшей ее транзакцией.

RECORD_VERSION читает самую последнюю подтвержденную (commit) версию строки, даже если есть более поздняя, но неподтвержденная версия.

Конструкция RESERVING дает возможность транзакции регистрировать желаемый уровень доступа для указанных таблиц при старте транзакции, а не тогда, когда транзакция начинает фактически работать с таблицей. Резервирование таблиц при старте транзакции может позволить избежать возникновения тупиковых ситуаций из-за блокировки данных.

Предложение USING, доступное только в SQL, можно использовать для экономии системных ресурсов, ограничивая число баз данных, к которым транзакция может обращаться.

Другая внедренная команда SQL устанавливают заданную по умолчанию транзакцию с уровнем изоляции READ COMMITTED. Если тран-

закция сталкивается с конфликтом модификации, она ждет, чтобы получить управление, когда первая (блокирующая) транзакция зафиксирует результаты (commit) или произведет откат (rollback).

Пример 9.1

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

Следующая команда внедренного SQL стартует поименованную транзакцию:

Пример 9.2

```
EXEC SQL
SET TRANSACTION NAME T1 READ COMMITTED;
```

Наконец, еще одна команда внедренного SQL стартует поименованную транзакцию и резервирует три таблицы:

Пример 9.3

```
EXEC SQL
SET TRANSACTION NAME TR1
ISOLATION LEVEL READ COMMITTED
NO RECORD_VERSION WAIT
RESERVING TABLE1, TABLE2 FOR SHARED WRITE,
TABLE3 FOR PROTECTED WRITE;
```

Глава 10

Разработка приложений для работы с InterBase

Разработка приложений для работы с InterBase может осуществляться на различных языках программирования в соответствии с потребностями конкретного приложения и вкусами и привычками разработчика. Доступ к базе данных реализуется на основе вызова функций API. На всех поддерживаемых InterBase платформах можно использовать языки C и C++. Кроме того, на ряде платформ поддерживаются языки Ada (Alsys), Ada (VERDIX, VMS, Telesoft), ANSI-85 COBOL, COBOL, FORTRAN, Pascal.

Непосредственный вызов функций API требует предварительной подготовки текстов команд SQL, подготовки специальных структур для передачи данных от приложения к InterBase и обратно. Все это не только достаточно трудоемко, но и делает программы практически непереносимыми.

Для решения этой проблемы в состав InterBase включена утилита gpre, которая обеспечивает препроцессорную обработку текстов программ. В этом случае в текст программ просто вносятся соответствующим образом оформленные тексты на SQL. Тексты SQL внутри программы (внедренный SQL) начинаются с волшебных слов EXEC SQL, за которыми следует сама команда и которые нужны препроцессору для отделения SQL текста от текста программы на базовом языке. Утилита gpre обрабатывает текст и производит замену этих текстов на вызовы функций API. Порядок подготовки программ к обработке gpre рассматривается ниже.

Помимо этого такие системы, как C++ Builder и Delphi для Windows имеют специальные средства обеспечения доступа к базам данных. Эти средства включают как независимые от типа базы данных, так и непосредственно ориентированные на работу с InterBase.

10.1. Разработка приложений на базовом языке

Общие требования к приложениям

Все внедренные приложения должны включать некоторые объявления и команды, чтобы гарантировать надлежащую обработку препроцессором InterBase (GPRE) и обеспечить необходимую связь между SQL и базовым языком, на котором реализовано приложение. Каждое приложение должно:

- Объявлять базовые (host) переменные для передачи данных между SQL и приложением.
- Объявлять и устанавливать связь с базами данных, к которым обращается программа.
- Создать дескрипторы транзакций для каждой транзакции, отличной от транзакции по умолчанию, используемой в программе.
- Включать SQL и, возможно, DSQL команды.
- Обеспечивать обработку ошибок и восстановление.
- Закрывать все транзакции и базы данных перед окончанием программы.

При работе приложения могут использовать средства как статического, так и динамического SQL.

Напомню, что динамические приложения SQL это такие приложения, которые формируют команды SQL во время выполнения или дают возможность пользователям формировать их. К динамическим приложениям SQL предъявляются дополнительные требования.

Объявление базовых (ведущих) переменных

Ведущая переменная - стандартная переменная базового языка для чтения значений из базы данных, подготовки значений, для записи в базу данных или хранения значений, описывающих условия поиска в базе данных. SQL использует эти переменные в следующих целях:

- При поиске данных SQL помещает значения из полей базы данных в базовые переменные, где они могут просматриваться и изменяться.
- При запросе у пользователя информации базовые переменные используются для ее хранения и последующего использования InterBase в командах SQL INSERT или UPDATE.

- При задании условий поиска в команде SELECT условия могут быть указаны либо непосредственно, либо в базовой переменной. Например, следующие фрагменты команды SQL задают правильные предложения WHERE: первый - непосредственно, второй, используя переменную базового языка, wBook, для сравнения со столбцом BOOKNM.

Пример 10.1

```
EXEC SQL
 SELECT * FROM TBOOK WHERE BOOKNM = "Word 6 for Windows";
EXEC SQL
 SELECT * FROM TBOOK WHERE BOOKNM = :wBook;
```

Для каждого используемого столбца данных в базе должна быть объявлена ведущая (базовая) переменная. Через эти переменные и осуществляется обмен информацией между программой и базой данных. Сами ведущие переменные, используемые в программах SQL, объявляются точно так же, как стандартные переменные языка. Они следуют всем стандартными правилами для объявления, инициализации и использования. Например, в С эти требования сводятся к тому, что переменные должны быть объявлены прежде, чем они могут использоваться как базовые в командах SQL.

Пример 10.2

```
int wUNIKEY, wMOTHERKEY;
char wBook [251];
```

В ряде других систем (не InterBase) на порядок объявления базовых переменных налагаются дополнительные требования, а именно: они должны объявляться внутри блока BEGIN DECLARE SECTION и END DECLARE SECTION.

В связи с этим в целях обеспечения переносимости InterBase поддерживается раздел объявлений со следующим синтаксисом:

```
EXEC SQL
BEGIN DECLARE SECTION;
<hostvar_decl>;
. . .
EXEC SQL
END DECLARE SECTION;
```

<hostvar_decl> - объявление переменной на базовом языке.

Если при разработке приложения предполагается, что оно в дальнейшем может быть использовано и для другой СУБД, то для упрощения

действий по переносу все ведущие (базовые) переменные лучше объявлять между командами BEGIN DECLARE SECTION и END DECLARE SECTION.

В следующем примере ведущие переменные (см. пример 10.2) декларируются в пределах раздела объявлений:

Пример 10.3

```
EXEC SQL
BEGIN DECLARE SECTION;
 int wUNIKEY, wMOTHERKEY;
 char wBook [251];
EXEC SQL
END DECLARE SECTION;
```

Требование на объявление переменных внутри DECLARE SECTION распространяется в любом случае только на те переменные, которые используются в командах SQL, прочие переменные могут быть объявлены вне DECLARE SECTION везде, где это допускается синтаксисом базового языка.

С помощью подобных объявлений можно обеспечить связь с любыми столбцами базы данных, однако, в случае каких-либо изменений в структуре столбцов, необходимо будет везде в программе внести соответствующие изменения. При работе с большими программными комплексами процедура внесения таких изменений становится весьма трудоемкой и чреватой ошибками, часть из которых может проявиться, что особенно неприятно, далеко не сразу.

Для решения этой проблемы при объявлении переменных предусмотрен механизм конструкций BASED ON.

В InterBase поддерживается декларативное предложение BASED ON для создания переменных языка С, основанных на определениях столбца базы данных, прежде всего символьных, размерные характеристики которых наиболее подвержены изменениям. Препроцессор по описанию столбца базы данных самостоятельно определяет описание переменных. Использование BASED ON гарантирует, что результирующая переменная базового языка будет достаточно велика, чтобы содержать максимальное число символов столбца базы данных в формате CHAR или VARCHAR плюс дополнительный байт '\0' для конечного ограничителя, ожидаемых большинством функций С, работающих со строками. BASED ON использует синтаксис

```
BASED ON <dbcolumn> <hostvar>;

<dbcolumn>::=[<dbh>.]<table>.<column>
  <dbh>    - указатель базы данных
```

<table> - имя таблицы базы данных
<column> - имя столбца базы данных

<hostvar> - имя базовой переменной, используемой для размещения данных соответствующего столбца

В командах следующего примера ведущие переменные объявляются на основании столбцов таблицы TBOOK нашей базы.

Пример 10.4

```
BASED ON TBOOK.UNIKEY wUNIKEY;
BASED ON TBOOK.MOTHERKEY wMOTHERKEY;
BASED ON TBOOK.BOOKNM wBook;
```

В результате обработки препроцессором программы на C или C++ эти объявления этих переменных примут вид:

```
int wUNIKEY;
int wMOTHERKEY;
char wBook [251];
```

Перед использованием конструкции BASED ON необходимо задать базу, с которой будет выполняться работа, выполнить подключение к этой базе (после этого препроцессору станут доступны описания столбцов), задать секцию объявления переменных и только затем сделать само объявление. Порядок подключения к базам данных будет рассмотрен чуть дальше, а пока рассмотрим использование конструкции BASED ON на примере, включающем необходимые предварительные действия. Эти действия реализуются следующей последовательностью команд:

- С помощью SET DATABASE задается база данных, из которой берутся определения столбцов.
- Выполняется команда CONNECT, для соединения с базой данных.
- Командой BEGIN DECLARE SECTION задается раздел объявлений.
- Командой BASED ON объявляются переменные.

В приведенном ниже примере с помощью BASED ON объявлена одна строковая переменная. Целые переменные объявлены прямо без использования конструкции BASED ON.

Пример 10.5

```
EXEC SQL
SET DATABASE MYBASE = "testbase.gdb";
```

```
EXEC SQL
CONNECT MYBASE;
EXEC SQL
BEGIN DECLARE SECTION;
int wUNIKEY;
BASED ON MYBASE.TBOOK.BOOKNM wBook;
EXEC SQL
END DECLARE SECTION;
```

Обработку строк данных, выбираемых из базы удобнее выполнять, если данные одной строки сгруппированы вместе. В С для этого можно использовать структуры. Элементы структур допустимы для использования в качестве базовых переменных в командах SQL.

Например, для доступа к таблице TBOOK нашей базы естественно использовать структуру, приведенную в следующем примере.

Пример 10.6

```
struct Data_book
{int wPRMKEY;
 int wMATHERKEY;
 char wBOOKNM[251];
};

Data_book book, book_array[50];
```

Использование объявленной структуры для чтения данных из базы в этом случае можно проиллюстрировать следующим примером.

Пример 10.7

```
EXEC SQL
SELECT PRMKEY, MATHERKEY, STREET, BOOKNM
INTO : book.wPRMKEY, : book.wMATHERKEY,
: book.wBOOKNM
FROM TBOOK WHERE MATHERKEY > 0;
```

Условие в данном примере предназначено для отсечения наименований рубрик, поскольку в таблице хранятся и названия книг и рубрики. Поле MATHERKEY указывает на родительскую рубрику, а рубрика не имеет родителя. Если рубрикатор многоуровневый, то такое условие недостаточно, но для иллюстрации использования структур в качестве базовых переменных это не существенно.

Объявление баз данных и соединение с базами данных

Для того чтобы программа могла работать с базой данных, эта база должна быть объявлена. Объявление может быть сделано либо явно, либо косвенно путем задания соответствующего объявление в параметрах утилиты GPRE. Краткое описание работы с препроцессором GPRE приведено ниже. Если программа работает с несколькими базами, то явных объявлений избежать нельзя.

Отметим, что программы, использующие DSQL, не могут работать одновременно с нескольким базам данных InterBase.

Для подключения к базе данных InterBase необходимо объявить базу, а затем выполнить соединение с ней. Выполнение этих действий реализуется следующими командами SQL:

- SET DATABASE объявляет имя базы данных, с которой предполагается работать и назначает ей дескриптор базы данных. Дескриптор базы данных используется для идентификации базы данных во всех командах, работающих с базой. Наличие дескриптора позволяет абстрагироваться от физического расположения базы. С дескриптором связывается описание характеристик базы. Дескрипторы могут также использоваться, чтобы квалифицировать имена таблиц при работе с несколькими базами.
- CONNECT открывает базу данных, указанную дескриптором, и выделяет для нее системные ресурсы.

Команда SET DATABASE предназначена для:

- объявления дескрипторов для каждой из используемых в программе базы данных.
- связывавания дескриптора базы данных с фактическим именем базы данных. Дескриптор базы данных является, по существу, псевдонимом (алиасом) базы данных в программе.

SET DATABASE объявляет и одновременно устанавливает ведущую (базовую) переменную для дескриптора базы. Дескриптор базы данных содержит указатель, используемый для ссылки на базу данных в последующих командах SQL. Для включения команды SET DATABASE в программе используется следующий синтаксис:

```
EXEC SQL
SET DATABASE dbhandle = "<dbname>";
```

dbhandle - имя дескриптора базы данных **в программе**
<dbname> - имя базы данных

Полный синтаксис команды, приведенный ниже, несколько сложнее и позволяет помимо создания дескриптора указать имя пользователя, пароль, имя роли.

```
SET DATABASE dbhandle = [GLOBAL / STATIC / EXTERN]
[COMPILETIME] [FILENAME] 'dbname'
[USER 'username' PASSWORD 'password']
[RUNTIME [FILENAME] {'dbname' / :vardbn}
[USER {'username' / :varnm} PASSWORD {'password' /
:varpas}] ];
```

GLOBAL, STATIC, EXTERN задают режим объявления: для всех модулей, для данного или ссылаются на сделанное в другом месте объявление.

COMPILETIME, RUNTIME задают «для кого» сделано объявление. COMPILETIME - для препроцессора GPRE, чтобы обеспечить настройку и проверку команд SQL в программе. RUNTIME - собственно для рабочей программы.

dbhandle - имя дескриптора базы данных в программе

dbname - имя базы данных, вообще говоря, с полным путем доступа к ней. Для режима RUNTIME вместо литеральной строки можно использовать базовую переменную.

Конструкция USER 'username' PASSWORD 'password' задает имя пользователя и пароль. Для режима RUNTIME вместо литеральных строк можно использовать базовые переменные.

В соответствии с синтаксисом команды для обеспечения работы с несколькими базами, необходимо выдать столько команд SET DATABASE, сколько предполагается использовать баз данных.

В качестве примера приведем следующую команду соединения с базой testbase.gdb (см. также пример 10.5).

Пример 10.8

```
EXEC SQL
SET DATABASE MYBASE = "testbase.gdb";
```

Создание дескриптора не означает автоматического соединения с базой данных. Для физического соединения необходимо выдать команду CONNECT. При работе с одной базой в локальной сети можно соединяться с базой сразу же после создания дескриптора соединяться с базой. Если же работа ведется с несколькими базами, особенно с удаленными, то для уменьшения нагрузки сети есть смысл соединяться с ними только непосредственно перед началом работы и отсоединяться сразу же по ее окончании.

Отсоединение от базы осуществляется командой DISCONNECT.

Итак, команда CONNECT обеспечивает соединение с базой данных (открывает базу данных) и выделяет для нее системные ресурсы. После открытия базы данных можно использовать таблицы, процедуры и другие объекты базы данных, доступ к которым разрешен пользователю при соединении. Команда CONNECT используют следующий основной синтаксис

```
CONNECT [TO] LIST_<connect>

<connect>::= <db_specs> <config_opts>

<db_specs>::= dbhandle
  / {' dbname' / :variable} AS dbhandle

<config_opts>; := [USER {'username' / :variable}]
        /PASSWORD {'password' / :variable}]
        /ROLE {'rolename' / :variable}]
        [CACHE int [BUFFERS]]
```

dbhandle - имя дескриптора базы данных в программе

dbname - имя базы данных, вообще говоря, с полным путем доступа к ней. Вместо литеральной строки можно использовать базовую переменную.

Конструкция USER 'username' PASSWORD 'password' задает имя пользователя и пароль. Вместо литеральных строк можно использовать базовые переменные.

Конструкция CACHE - BUFFERS задает количество буферов кэша, которое определяет количество страниц базы, доступных одновременно. В большинстве случаев можно не задавать явно, используя значения по умолчанию.

Из синтаксиса команды видно, что ряд характеристик соединения можно задавать в нескольких местах: в командной строке препроцессора, в команде SET DATABASE, в команде CONNECT. Важно лишь, чтобы все обязательные параметры соединения были заданы.

И еще одно замечание. С помощью одной команды CONNECT можно соединиться с несколькими базами данных.

Приведем примеры соединения с базой данных.

Пример 10.9

```
EXEC SQL
CONNECT MYBASE;
```

Здесь мы связываемся с той базой, которая была установлена для дескриптора MYBASE. Если считать, что для установки дескриптора использовалась команда предыдущего примера, то это будет "testbase.gdb".

Пример 10.10

```
EXEC SQL
CONNECT MYBASE USER 'SYSDBA' PASSWORD 'masterkey';
```

Здесь явно указывается пользователь и его пароль.

Отдельная команда CONNECT может использоваться для соединения с каждой базой данных, либо одна команда может соединяться с несколькими базами данных. Рассмотрим оба варианта, считая что соответствующие дескрипторы баз были ранее уже установлены:

Пример 10.11

```
EXEC SQL
CONNECT DataBase!.;
EXEC SQL
CONNECT DataBase2 USER 'MISHA' PASSWORD 'ahsim';
```

либо

```
EXEC SQL
CONNECT DataBase!.,
        DataBase2 USER 'MISHA' PASSWORD 'ahsim';
```

А следующий пример использует одну команду CONNECT, чтобы подключиться к двум базам:

Как только база данных подключена, к ее таблицам можно обращаться в последующих транзакциях. Ее дескриптор может квалифицировать имена таблицы в приложениях SQL, но не в приложениях DSQL.

Для разрыва соединения с базой данных следует использовать команду DISCONNECT. Команда имеет следующий синтаксис

```
DISCONNECT {{ALL | DEFAULT} | LIST_dbhandle};
```

dbhandle - имя дескриптора базы данных в программе.

Конструкция ALL выполняет разрыв соединения со всеми базами, с которыми было выполнено соединение.

Конструкция DEFAULT выполняет разрыв соединения с принятой по умолчанию (единственной) базой данных.

Замечание 1. При использовании препроцессора GPRE соединение с базой можно задать соответствующими параметрами в командной стро-

ке. Тогда в программе, работающей с одиночной базой данных и обработанной препроцессором GPRE без ключа -т (ключ -т подавляет автоматическое порождение транзакций), команды SET DATABASE и CONNECT необязательны. В то же время нет никаких разумных оснований от отказа использования в теле программы команд SET DATABASE и CONNECT. В этом случае программа оказывается зависимой от параметров препроцессора, кроме того, возникает необходимость в дополнительном документировании программы. Если эти команды все же опущены, необходимо выполнить следующие шаги:

1. Вставить в код программы раздел объявлений, где определяются глобальные переменные. Если никакие переменные базового языка не используются в программе, следует использовать пустой раздел объявлений.

Пример 10.12

Пустой раздел объявлений.

```
EXEC SQL
BEGIN DECLARE SECTION;
EXEC SQL
END DECLARE SECTION;
```

2. Определить имя базы данных в командной строке GPRE.

Замечание 2. База данных не должна задаваться, если программа содержит команду CREATE DATABASE.

Работа с транзакциями

Все команды SQL явно или неявно выполняются в рамках транзакций. Таким образом, перед обращением к базе необходимо объявить транзакцию. После ее объявления следующие команды SQL будут по умолчанию относится именно к этой транзакции.

Можно также объявить несколько транзакций, которые будут существовать одновременно. В этом случае при выдаче команд SQL необходимо явно указывать в рамках какой именно транзакции они выполняются. Для обеспечения подобных ссылок транзакции могут быть поименованы.

Команда создания транзакции имеет следующий синтаксис.

```
SET TRANSACTION [NAME transaction]
[READ WRITE | READ ONLY] [WAIT | NO WAIT]
[[ISOLATION LEVEL] {SNAPSHOT |TABLE STABILITY.|
| READ COMMITTED [[NO] RECORD_VERSION]}]
[RESERVING LIST_<table_d> | USING LIST_dbhandle];
```

```
<table_d>::= table_name [FOR [SHARED / PROTECTED] {READ
/ WRITE}]
```

Конструкция NAME transaction задает необязательное имя транзакции.

Конструкция READ WRITE, READ ONLY разрешает чтение и запись в базу или только чтение, если отсутствует, то принимается READ WRITE.

Конструкция ISOLATION LEVEL задает один из допустимых в InterBase уровней изоляции (см. предыдущий раздел). По умолчанию задается уровень SNAPSHOT.

Конструкция RESERVING LIST_table задает список таблиц блокируемых транзакцией и режимов блокировки. Этот режим полезен при большом объеме обновлений в таблицах, выполняемых в течение короткого времени, иначе это может «подвесить» всех пользователей базы.

Конструкция USING LIST_dbhandle ограничивает доступ к базе данных перечисленными дескрипторами баз данных.

Рассмотрим объявление транзакций на следующих примерах.

Объявим неименованную транзакцию с уровнем изоляции SNAPSHOT.

Пример 10.13

```
EXEC SQL
SET TRANSACTION;
```

Поскольку этот уровень принимается по умолчанию то его можно не указывать явно. Полное описание этой транзакции с учетом значений, принимаемых по умолчанию будет.

```
EXEC SQL
SET TRANSACTION READ WRITE WAIT ISOLATION LEVEL SNAPSHOT;
```

Объявим теперь поименованную транзакцию с уровнем изоляции READCOMMITTED.

Пример 10.14

```
EXEC SQL
        BEGIN DECLARE SECTION;
long  *trans1;

. . .

EXEC SQL
```

```
SET TRANSACTION NAME transl READ WRITE WAIT ISOLATION LEVEL
READ COMMITTED;
```

После выполнения набора действий с базой данных они должны быть либо зафиксированы, либо отменены. Окончание такого набора действий фиксируется завершением транзакции. Фиксация результатов производится командой COMMIT, отмена - ROLLBACK. Отказ от закрытия транзакции до окончания программы может породить транзакцию с неопределенным состоянием (in limbo), когда записи внесены в базу данных, но ни подтверждены, ни отменены. Транзакции с неопределенным состоянием могут быть очищены, только используя административные средства базы данных, входящие в состав InterBase.

Команда COMMIT фиксирует изменения сделанные в базе командами, выполненными между SET TRANSACTION и COMMIT. Команда COMMIT имеет следующий синтаксис

```
COMMIT [WORK] [TRANSACTION name] [RETAIN [SNAPSHOT]];
```

WORK - необязательное слово, используемое для унификации синтаксиса SQL, используемого для разных СУБД.

TRANSACTION name используется для идентификации именованной транзакции.

Конструкция RETAIN [SNAPSHOT] фиксирует изменения в базе данных и открывает новую транзакцию с тем же именем. Таким образом, новая транзакция, оказывается, по существу «старой» (текущее состоянии транзакций TIP не меняется, см. механизм работы с транзакциями). В многопользовательской среде сохранение текущего состоянии (снимка) ускоряет обработку и требует меньшего количество ресурсов системы по сравнению с закрытием и стартом новой транзакции для каждого действия. Недостатком использования RETAIN [SNAPSHOT] является то, что до полного закрытия транзакции (без конструкции RETAIN) нельзя видеть изменения, сделанные другими транзакциями.

В качестве примера рассмотрим закрытие транзакций, созданных в примерах 10.13, 10.14.

Пример 10.15

```
EXEC SQL

SET TRANSACTION;

COMMIT RETAIN;

COMMIT;
```

В данном примере сначала сохраняются сделанные изменения, сделанные группой команд между SET TRANSACTION и COMMIT RETAIN, а затем выполняется окончательное завершение транзакции.

Пример 10.16

```
EXEC SQL
SET TRANSACTION NAME trans1 READ WRITE WAIT ISOLATION LEVEL
READ COMMITTED;
. . .
COMMIT transl;
```

Здесь завершается **одна** из возможно нескольких активных транзакций по ее имени.

В случае возникновения каких-либо ошибок результаты действий по внесению изменений в базу должны быть отменены. Для этого используется команда ROLLBACK.

Команда ROLLBACK имеет следующий синтаксис.

```
ROLLBACK [TRANSACTION name] [WORK] [RELEASE];
```

WORK - необязательное слово, используемое для унификации синтаксиса SQL, используемого для разных СУБД.

TRANSACTION name используется для идентификации именованной транзакции.

RELEASE выполняет отсоединение от базы данных. Данную опцию трудно рекомендовать к применению. Для отсоединения от базы лучше использовать команду DISCONNECT.

В качестве примера возьмем модификацию предыдущего примера.

Пример 10.17

```
EXEC SQL
SET TRANSACTION NAME transl READ WRITE WAIT ISOLATION LEVEL
READ COMMITTED;
. . .
ROLLBACK transl;
```

Использование команд SQL. Обработка ошибок.

Включение в текст приложения команд SQL не вызывает каких-либо трудностей. Единственное о чем нельзя забывать, так это о добавлении к команде ключевых слов EXEC **SQL.**

Кроме того, необходимо помнить об особенностях применения команды SELECT.

Для команды SELECT, возвращающей в точности одну строку применим синтаксис

```
EXEC SQL
SELECT [TRANSACTION transaction] [DISTINCT / ALL]
       {* | <val> [, <val> ...]} INTO :var [, :var ...] ]
...;
```

ключевым элементом которого в интересующем нас смысле является конструкция INTO, задающая список базовых переменных, в которые помещаются результаты выборки.

Для получения многострочной выборки необходимо использовать курсоры.

Работа с курсором включает три этапа

- Объявление курсора.
- Открытие курсора.
- Получение очередной строки выборки.

Объявление курсора имеет следующий вид:

```
EXEC SQL
DECLARE cursorname CURSOR FOR
SELECT ...;
```

cursomame задает имя курсора.

После ключевого слова SELECT могут использоваться любые конструкции, допустимые для команды SELECT, кроме конструкции INTO, поскольку в данном случае никакого считывания данных производиться просто не может.

Команда открытия курсора имеет следующий синтаксис.

```
EXEC SQL
OPEN cursorname;
```

cursorname задает имя ранее объявленного курсора.

По команде OPEN cursor выполняется собственно обращение к базе, после которого можно непосредственно считывать данные командой FETCH.

Команда FETCH имеет следующий синтаксис.

```
EXEC SQL
FETCH cursorname INTO LIST_<var>;
```

```
<var>::= :variable [[INDICATOR] : indvar]
```

cursomame задает имя ранее открытого курсора.

variable - имя базовой переменной, в которую помещается значение считанного столбца.

indvar - задает имя переменной, в которую помещается признак того, что считанное значение поля - NULL. Формат поля - **short,** при получении значения NULL для выбираемого поля индикатор устанавливается в –1, иначе 0.

Таким образом, схема чтения данных из базы приобретает следующий вид.

Пример 10.18

```
EXEC SQL
DECLARE Mcursor CURSOR FOR
SELECT ...;
. . .

EXEC SQL
OPEN Mcursor;
. . .
// Открытие цикла чтения
. . .
EXEC SQL                                                    :
FETCH Mcursor INTO ...;
// Проверка на конец выборки и выход из цикла
. . .
// Окончание цикла чтения
```

Теперь несколько слов об обработке событий, связанных с получением данных из базы и запись в базу.

При выполнении любой команды SQL в переменную SQLCODE помещается код ошибки. Сама переменная SQLCODE объявляется автоматически по результатам обработки препроцессором gpre.

Итак, в нашем примере после команды FETCH следует поставить `if(SQLCODE) break;`

Это обеспечит выход из цикла после исчерпания списка строк.

Аналогичная конструкция должна использоваться, вообще говоря, после всех SQL команд для обеспечения корректной обработки возникающих ошибок.

Работа с DSQL

Теперь несколько слов о проблемах, возникающих при работе с динамическим SQL (DSQL). Приложения DSQL в отличие от обычных дают возможность пользователям вводить специальные команды SQL для обработки во время выполнения.

Приложения, использующие DSQL должны жестко придерживаться всех требований для приложений SQL и, кроме того, удовлетворять ряду дополнительных требований.

Поскольку команды должны обрабатываться в процессе выполнения, то необходимо явно связывать с каждой командой области памяти, в которых будут размещаться их входные и выходные параметры. Для организации такой связи используется блок специальной структуры - расширенная область описателя SQL (XSQLDA). XSQLDA используется, как промежуточное звено для передачи информации между приложением и InterBase, описывающее состав информации и содержащее указатели на место ее хранения. XSQLDA используется для следующих задач:

- Передачи параметров от базового языка к SQL.
- Приема параметров от команды SELECT или хранимой процедуры программой на базовом языке.
- В каждый момент времени XSQLDA может обслуживать только одну из этих задач. Большинство приложений объявляют два блока XSQLDA, один для ввода, и другого для вывода.

Структура XSQLDA определена в заголовочном файле InterBase ibase.h, который автоматически включен в программы, при из обработке препроцессором gpre.

Чтобы обрабатывать различные команды DSQL, которые может вводить пользователь, при программировании требуются следующие дополнительные шаги:

- Объявляются расширенные области описателя SQL (XSQLDA), обычно программа должна использовать одну или две такие структуры. В сложных приложениях может потребоваться и большее количество.
- Объявляются имена всех транзакций и дескрипторы баз данных, используемых в программе. Имена и дескрипторы определяются статически, поэтому они должны быть известны во время компиляции; объявлены они должны быть так, чтобы покрыть все ожидаемые потребности пользователей во время выполнения.

- Программируется механизм получения команд SQL от пользователя. Обычно это не сами команды, а исходные данные для них получаемые в диалоге, на основе которых сама программа и строит необходимые команды SQL. В ряде случаев генерация SQL может потребоваться и просто при решении сложных задач обработки данных даже без всякого участия пользователя.
- Подготавливаются все команды SQL, получаемые для обработки от пользователя. Командой PREPARE загружают данные команды SQL в XSQLDA. Каждую подготовленную инструкцию выполняют командой EXECUTE. Для объединения команд PREPARE и EXECUTE в одной можно использовать команду EXECUTE IMMEDIATE.

ОБЪЯВЛЕНИЕ XSQLDA

Блок XSQLDA предназначен для обеспечения передачи и приема параметров запросов команд SQL. Структура блока представлена на следующем рисунке

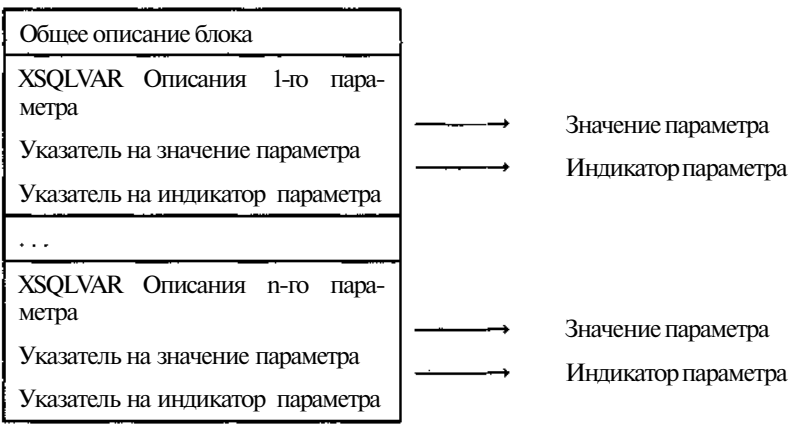


Рис. 10.1. Структура блока XSQLDA

Блок XSQLDA объявляется в секции объявлений. Проиллюстрируем это следующим примером.

Пример 10.18

```
EXEC SQL
```

```
BEGIN DECLARE SECTION;
XSQLDA in_sqlda;
XSQLDA out_sqlda;
. . .
EXEC SQL
END DECLARE SECTION;
```

Здесь объявляются два блока XSQLDA: in_sqlda для ввода данных из базы, out_sqlda - для вывода. Сразу заметим, что количество объявлений может быть любым. Важно лишь чтобы их было достаточно для параллельно исполняемых команд.

Следующие пункты продекларированной программы действий не имеют прямого отношения к работе с базой данных и сильно зависят от конкретной решаемой задачи, так что здесь достаточно рассмотреть только последний этап, заключающий подготовку запроса к базе.

ПОДГОТОВКИ ЗАПРОСА К ВЫПОЛНЕНИЮ

Команда подготовки запроса к выполнению PREPARE имеет следующий синтаксис.

PREPARE [TRANSACTION transaction] statement
[INTO SQL DESCRIPTOR xsqlda] **FROM** {:variable | 'string'};

TRANSACTION transaction задает имя транзакции, в рамках которой будет выполняться команда (если опущено, то заданной по умолчанию).

statement - устанавливает псевдоним для подготовленной команды. Псевдоним в дальнейшем используется командой EXCUTE для выполнения команды, заданной в текстом, указанным в конструкции FROM.

FROM {:variable | 'string'} задает текст SQL, который должен в дальнейшем исполняться. Текст может задаваться либо переменной, либо строковым литералом. Если запрос содержит параметры, то в тексте запроса на месте значений параметров указывается «?». Например, «UPDATE TBOOK SET bookmn = ? WHERE unikey = ? ;».

INTO SQL DESCRIPTOR xsqlda задает область XSQLDA, подготавливаемую для приема входных параметров.

ВЫПОЛНЕНИЕ ЗАПРОСА

Выполнение подготовленного текста осуществляется командой EXECUTE.

Команда EXCUTE имеет следующий синтаксис.

EXECUTE [TRANSACTION transaction] statement
[USING SQL DESCRIPTOR xsqlda] **[INTO SQL DESCRIPTOR xsqlda];**

TRANSACTION transaction задает имя транзакции, в рамках которой будет выполняться команда (если опущено, то заданной по умолчанию).

statement - псевдоним установленный командой PREPARE.

USING SQL DESCRIPTOR xsqlda задает используемый дескриптор блока XSQLDA для входных параметров.

INTO SQL DESCRIPTOR xsqlda задает дескриптор блока XSQLDA для размещения результатов работы.

Если входные или выходные параметры не используются, то соответствующие конструкции следует опустить. А вот если они используются, то необходимо предварительно выполнить следующие действия.

- Задать число параметров, выделить для них память в XSQLDA, если ранее выделенной памяти недостаточно. Размер памяти определяется размером фиксированной части XSQLDA плюс размером описателя параметра (размер структуры XSQLVAR) умноженным на количество параметров.
- Установить типы данных для каждого параметра.
- Установить указатели на области памяти, где будут размещены фактические значения для параметров.
- Установить указатели на области памяти, где будут размещены индикаторы для NULL значений.
- При желании можно также задать некоторые описатели данных для формирования диагностики, но это необязательно.

Далее работа может вестись так же, как и с обычными командами SQL.

Если по характеру задачи команда выполняется однократно и не имеет возвращаемых значений, то вместо пары команд PREPARE и EXECUTE можно использовать команду EXECUTE IMMEDIATE. Команда EXECUTE IMMEDIATE имеет следующий синтаксис.

```
EXECUTE IMMEDIATE [TRANSACTION transaction]
{:variable | 'string'} [USING SQL DESCRIPTOR xsqlda];
```

TRANSACTION transaction задает имя транзакции, в рамках которой будет выполняться команда (если опущено, то заданной по умолчанию).

{:variable | 'string'} задает текст SQL, который должен в дальнейшем исполняться. Текст может задаваться либо переменной, либо строковым литералом.

USING SQL DESCRIPTOR xsqlda задает область XSQLDA, подготовленную для входных параметров.

ОГРАНИЧЕНИЯ ПРИ РАБОТЕ С DSQL

DSQL дает возможность создать гибкие приложения, которые могут обрабатывать широкий спектр запросов пользователя. В то же время необходимо помнить, что не каждая команда SQL может быть обработана полностью динамическим способом. Например, дескрипторы базы данных и имена транзакций должны быть определены при написании приложения и не могут быть изменены или определены пользователями во время выполнения. Хотя InterBase и поддерживает одновременную работу с несколькими базами данных и несколькими транзакциями в одном приложении, существуют следующие ограничения на DSQL:

- Одновременно можно обращаться только к одной базе данных.
- Транзакции могут работать только с активной в настоящее время базой данных.
- Пользователи не могут определять имена транзакций в инструкциях DSQL. Имена транзакций должны быть заданы заранее и быть известны на момент компиляции.

ИСПОЛЬЗОВАНИЕ ДЕСКРИПТОРОВ БАЗЫ ДАННЫХ

Дескрипторы базы данных всегда объявляются статически, поэтому количество дескрипторов должно быть достаточным для удовлетворения ожидаемых потребностей пользователей. Но если дескриптор объявлен, то во время выполнения ему может быть с помощью команды SET DATABASE назначена любая указанная пользователем база данных, как, например, в следующем фрагменте кода C:

Пример 10.19

```
. . .
EXEC SQL
SET DATABASE DB = "dummydb.gdb";
. . .
// Вводим в переменную Db_Param имя базы данных
. . .

EXEC SQL
SET DATABASE DB = :Db_Param;
. . .
```

ИСПОЛЬЗОВАНИЕ АКТИВНОЙ БАЗЫ ДАННЫХ

Приложение DSQL может одновременно работать только с одной базой данных, даже если приложение присоединяется к нескольким базам данных. Из множества баз, соединенных с приложением, активной считается последняя база данных, связанная с дескриптором в команде SET DATABASE и именно с ней и будут работать команды DSQL.

Например, все команды DSQL, введенные пользователем во время выполнения, будут работать только с одной базой данных, но приложение может также содержать не DSQL-команды, которые делают запись в другие базы, используя, в том числе и данные вводимые пользователем.

ИСПОЛЬЗОВАНИЕ ИМЕН ТРАНЗАКЦИЙ

Многие команды SQL поддерживают необязательный параметр - имя, используемое для задания транзакции, в рамках которой выполняется данная команда. Имена транзакций могут использоваться также в приложениях DSQL, но они должны быть установлены перед компиляцией приложения. Как только имя объявлено, оно может быть непосредственно вставлено в команду пользователя, правда, лишь самим приложением.

После объявления можно использовать имя транзакции в команде EXECUTE или команде EXECUTE IMMEDIATE.

ПЕРЕНОС ПРИЛОЖЕНИЙ ДЛЯ SQL

Принципиально существует возможность переносить приложения с текстами на внедренном SQL, разработанные для одной СУБД в другую.

При перенесении таких приложений в InterBase необходимо выполнение ряда условий. Например, многие варианты SQL требуют, чтобы host переменные были объявлены между BEGIN DECLARE SECTION и END DECLARE SECTION; InterBase не предъявляет таких требований, но препроцессор InterBase GPRE может корректно обрабатывать раздел объявлений переносимого приложения. Для дополнительной мобильности лучше объявлять все переменные базового языка в пределах соответствующих разделов.

При перенесении в InterBase существующих приложений DSQL, использующих другую область описателя SQL, они должны быть модифицированы так, чтобы разместить расширенный SQLDA (XSQLDA), используемый InterBase.

Правда, следует сразу предостеречь от излишнего оптимизма в части легкости подобного переноса.

- Внедренный SQL в разных СУБД, конечно похож, но полной совместимости, безусловно, нет.

- Если человек решил писать с использованием внедренного SQL, то есть на максимально приближенном к СУБД уровне, то он будет стараться максимально использовать ее специфические возможности для повышения производительности, а это означает и потерю совместимости.

Таким образом, не следует ожидать, что переход от одной системы к другой будет состоять просто в обработке исходного текста препроцессором и последующей перекомпиляции. Надо ясно понимать, что при выборе программирования с прямым использованием внедренного SQL проблема переноса будет трудноразрешимой.

Препроцессорная обработка программ

После того как программа SQL или DSQL написана, но до того как она откомпилирована и скомпонована, ее надо обработать препроцессором gpre. Препроцессор gpre транслирует команды SQL и переменные в команды и переменные, которые пригодны для компилятора базового языка, включая вызовы библиотечных функций InterBase. GPRE транслирует SQL- и DSQL-переменные базы данных в переменные базового языка, компилятор базового языка принимает и обрабатывает их. GPRE также объявляет некоторые переменные и структуры данных, требуемые SQL (типа SQLCODE-переменной и расширенной области описателя SQL (XSQLDA), используемой DSQL).

ИСПОЛЬЗОВАНИЕ GPRE

Синтаксис:

```
gpre [ -language] [ -options] infile [ outfile]
```

Параметр Infile определяет имя входного файла. Необязательный параметр outfile определяет имя выходного файла. Если он не определен, gpre создает выходной файл с тем же именем, как у входного, и расширением, зависящим от языка входного файла.

Gpre имеет ключи, которые задают язык исходной программы и ряд других опций. Ключи можете размещать как перед, так и после спецификации входного и выходного файлов. Каждый ключ должен включать, по крайней мере, дефис, которому предшествует пробел и уникальный символ, определяющий ключ.

ПЕРЕКЛЮЧАТЕЛИ ЯЗЫКА

Переключатель языка определяет язык исходной программы. Языки C и C++ доступны на всех платформах, и переключатели для них имеют вид:

-c C
-cxx C++

Кроме того, некоторые платформы поддерживают другие языки, если имеется соответствующая лицензия InterBase для языка:

-al[sys] Ada (Alsys)
-a[da] Ada (VERDIX, VMS, Telesoft)
-ansi ANSI-85 COBOL
-co[bol] COBOL
-f[ortran] FORTRAN
-pa[scal] Pascal

Например, для обработки препроцессором программы census.e, написанной на языке C, командная строка будет иметь вид

```
Gpre -c census.e
```

КЛЮЧИ ОПЦИЙ

Переключатели опций задают режимы предварительной обработки. Доступные переключатели описаны в табл. 10.1.

Таблица 10.1. *Переключатели утилиты gpre*

Переключатель	*Описание*
-charset name	Определяет активный набор символов во время компиляции, где name - имя набора символов
-d[atabase] filename	Объявляет базу данных для программ. filename - имя файла базы данных. Опция используется, если программа содержит команды SQL и не присоединяется к базе данных непосредственно. Не используется, если программа включает объявление базы данных
-d_float	Только для VAX/VMS. Определяет, что данные с двойной точностью будут приниматься из приложения в формате D_FLOAT, а сохраняться в базе данных в формате G_FLOAT. Сравнение данных в пределах базы будут выполняться в формате G_FLOAT. Данные, возвращенные приложению из базы данных, будут в формате D_FLOAT

Переключатель	***Описание***
-e[ither_case]	Указывает GPRE на необходимость различать верхний и нижний регистры. Переключатель используется, когда ключевые слова SQL появляются в коде в символах нижнего регистра. Если регистр смешан, а этот переключатель не используется, gpre не может обрабатывать входной файл. Этот переключатель необходим только с C, так как другие языки не различают регистр
-m[anual]	Подавляет автоматическое порождение транзакций. Используется для программ SQL, которые выполняют собственную обработку транзакций, и для всех программ DSQL, которые должны явно управлять собственными транзакциями по определению
-n[o_lines]	Подавляет номера строки для программ C
-o[utput]	Направляет вывод gpre на стандартный вывод, а не в файл
-password	Определяет пароль базы данных, если программа соединяется с базой данных, которая его требует
-r[aw]	Выводит BLR, как необработанные числа, а не как их мнемонические эквиваленты. Эта опции полезна для создания gpre меньшего выходного файла, однако, файл будет нечитабельным
-sqlda **[old | new]**	Параметр old определяет SQLDA, new определяет XSQLDA. Если этот переключатель не используется, то принимается значение по умолчанию - XSQLDA
-user**username**	Определяет username - имя пользователя базы данных, если программа соединяется с базой данных, которая его требует
-xhandle	Задает дескриптор базы данных, идентифицированный опцией -d[atabase], как внешнее объявление. Эта опция указывает программе, что глобальное объявление берется из другого связанного модуля. Используйте только совместно с переключателем -d[atabase]
-z	Выводит номер версии gpre и номера версий всех объявленных баз данных. Эти базы данных могут быть объявлены или в программе или в ключе -database

При наличии соответствующей лицензии и использовании языка, отличного от C, следует использовать дополнительные опции GPRE.

В следующем примере обрабатывается C-программа в файле appll.e. Выходным будет файл appll.c. Так как никакая база данных не определена, в исходном тексте должно быть предусмотрено соединение с базой данных.

Пример 10.20

```
gpre -c appl1
```

А этот пример аналогичен предыдущему, но не предполагает, что исходный текст открывает базу данных; вместо этого явно объявлена база данных, mydb.gdb.

Пример 10.21

```
gpre -c appl1 -d mydb.gdb
```

ИСПОЛЬЗОВАНИЕ РАСШИРЕНИЯ ФАЙЛА ДЛЯ ОПРЕДЕЛЕНИЯ ЯЗЫКА

В дополнение к использованию ключа для указания языка можно использовать просто расширение в имени исходного файла.

Таблица 10.2. *Расширения по умолчанию, используемые утилитой gpre*

Язык	*Расширение исходного файла*	*Расширение выходного файла*
Ada (VERDIX)	ea	a
Ada (Alsys, Telesoft)	eada	ada
C	e	c
C++	exx	cxx
COBOL	ecob	cob
FORTRAN	ef	f
Pascal	epas	pas

Например, для COBOL-программы census.ecob командная строка может иметь вид:

Пример 10.22

```
gpre census_report.ecob
```

Выходным будет файл census.cob.

КОМПИЛЯЦИЯ И КОМПОНОВКА

После предварительной обработки программа должна быть откомпилирована и скомпонована. Для компиляции используется компилятор базового языка.

Процесс компоновки разрешает внешние ссылки и создает выполнимый модуль. Особенности процесса компоновки зависят от используемой платформы.

10.2. Разработка приложений на C++ Builder и Delphi

Системы визуального программирования C++ Builder и Delphi содержат ряд компонент, специально ориентированных на работу с базами данных. Использование этих компонент позволяет быстро создавать приложения, работающие с базами данных. Кроме того, большая часть этих компонент построена таким образом, что обеспечивает максимальную переносимость программ, позволяя им работать (с минимальными изменениями, а в некоторых случаях вообще без изменений) с различными базами данных. Сразу, правда, следует оговориться, что данные системы ориентированы на работу под управлением Windows / Windows NT.

ОРГАНИЗАЦИЯ ДОСТУПА К ОБЪЕКТАМ БАЗЫ ДАННЫХ

Основными компонентами для доступа к объектам произвольных баз данных в C++ Builder и Delphi являются TDatabase, TSession, TTable и TQuery. Для работы с InterBase можно также использовать специализированные компоненты TIBTable, TIBQuery, TIBDatabase, TIBTransaction. Перечисленные компоненты составляют только часть возможных средств для работы с базами, но их достаточно для рассмотрения всех основных возможностей работы с базой данных.

Компоненты TDatabase, TTable и TQuery ориентированы на работу с произвольными базами данными, так что доступ к базам данных в них осуществляется не прямо, а через средства Borland Database Engine (BDE). Это является, с одной стороны, достоинством, обеспечивая переносимость программ, с другой - недостатком, поскольку часть возможностей InterBase, к счастью незначительная, оказывается недоступной.

Компоненты TIBTable, TIBQuery, TIBDatabase, TIBTransaction прямо ориентированы на работу с InterBase, обеспечивая реализацию всех его возможностей, но при этом более острой становится проблема переносимости программ для работы с другими СУБД.

Использование средств BDE при работе с InterBase

Прежде чем начать работу с базой данных, необходимо выполнить подключение к базе. Связь с базой реализуется объектом TDatabase.

Основные свойства TDatabase

AliasName Алиас базы данных, устанавливаемый средствами BDE Administrator. Алиас обеспечивает настройку BDE на работу с базой данных. Указание алиаса предназначено для настройки приложения па работу с конкретной базой.

DatabaseName Задает имя базы, на которое ссылаются компоненты работающие с базой. Обычно совпадает с алиасом.

Connected Указывает, что связь с базой данных, алиас которой задан свойством **AliasName,** установлена, если его значение есть **true,** или не установлена, если его значение - **false.** Установка свойства **Connected** в **true** эквивалентна выдаче SQL-команды CONNECT, установка свойства **Connected** в **false** эквивалентна выдаче SQL-команды DISCONNECT.

InTransaction Принимает значение **true,** если транзакция, связанная с базой данных активна, иначе - **false.**

При работе с BDE приложение может стартовать только одну транзакцию на каждую присоединенную базу данных. Однако для одной и той же базы можно установить несколько объектов **TDatabase,** имеющих один и тот же алиас **(AliasName),** но различные имена баз **(DatabaseName).** В этом случае они будут трактоваться, как разные базы и в каждой из них будет стартована своя транзакция. Такой, может быть, несколько искусственный прием позволяет иметь в приложении несколько параллельно работающих транзакций в одной базе.

TransIsolation Задает уровень изоляции для транзакций базы данных, управляемых BDE. При работе с BDE допустимы следующие три уровня: tiDirtyRead, tiReadCommitted и tiRepeatableRead. Поскольку InterBase уровень tiDirtyRead не поддерживает, то tiDirtyRead автоматически заменяется на tiReadCommitted. Реально возможны два уровня - tiReadCommitted и tiRepeatableRead, которым соответствуют уровни изоляции ReadCommitted и Snapshot в InterBase.

С одной базой данных можно связать много объектов типа TTable и TQuery, реализующих конкретные задачи, связанные с обработкой данных.

Наиболее важным для наших целей является объект TQuery.

Основные свойства TQuery

SQL Содержит текст команды на SQL, подлежащей выполнению.

Params Содержит список параметров запроса. При выполнении запроса значения параметров подставляются в выражение SQL, после чего

запрос компилируется. Это позволяет динамически формировать запросы, имитируя параметры, хотя сам SQL их и не поддерживает.

Prepared Признак готовности запроса к выполнению: **Prepared=true** - запрос подготовлен, **Prepared=false** - нет. Установка **Prepared=true** (или вызов метода Prepare()) вызывает компиляцию запроса; последнее существенно при многократно используемых запросах с параметрами.

DatabaseName Содержит имя базы данных, с которой работает запрос. Если объект TDatabase не создавался явно, то он будет создан по умолчанию со свойствами **AliasName** и **DatabaseName,** принимающими значение указанное в свойстве **DatabaseName** объекта TQuery. Если объекты TDatabase созданы явно (или по умолчанию, но ранее), то объект TQuery связывается с TDatabase по значениям полей **DatabaseName.** Свойство **AliasName** объекта TDatabase при этом может принимать другое значение.

Active Указывает, открыт **(Active=true)** или нет **(Active=false)** запрос. Применяется только к запросам, содержащим SQL-команду SELECT. Установка **Active** в **true** или **false** соответственно открывает или закрывает запрос (можно также использовать методы Open() и Close()).

Использование объектов TQuery, TDatabase при работе с InterBase

Сначала создаются сами объекты TQuery, TDatabase. Объект TDatabase может явно и не создаваться, в этом случае он будет все равно создан со свойствами по умолчанию.

Объект TQuery предназначен, прежде всего, для выполнения SQL-запросов к базе данных, поэтому в первую очередь необходимо связать с ним SQL-запрос. Текст запроса на SQL записывается в свойстве **SQL-**объекта.

Для выполнения запроса на выборку (команда SELECT) необходимо либо установить свойство объекта **Active** в **true,** либо вызвать метод Open().

После выполнения Open() доступ к результатам запроса осуществляется так, как если бы они были записаны в табличный файл. Переход от строки к строке осуществляется методами типа Next(), Prior(). Фактическая организация выборки приложению не видна. Соответственно, необходимости в командах работы с курсорами нет, более того, они вообще недоступны.

Для выполнения других команд SQL (текст их должен быть помещен в свойство SQL-объекта) необходимо вызвать метод ExecSQL().

Рассмотрим теперь, какие команды SQL доступны в приложении, использующем перечисленные объекты, а какие - нет. Недоступные для помещения в TQuery SQL-команды выделены курсивом.

Из приведенного перечня видно, что "потери" относятся почти исключительно к командам работы с курсорами, которые реализуются с точки зрения приложения несколько иначе и, на мой взгляд, удобнее. Единственные команды, которые остаются действительно недоступными — это команды обработки событий. Кроме того, управление транзакциями несколько ограничено, а именно, нельзя стартовать поименованную транзакцию. И еще, с базой данных в приложении может быть связана только одна транзакция. Последнее ограничение, впрочем, легко обходится объявлением нескольких объектов TDatabase с разными значениями DatabaseName и одинаковыми AliasName. Каждый из объектов имеет свою транзакцию, хотя физически они работают с одной базой.

При работе с данными отдельной таблицы можно использовать объект TTable. В этом случае просмотр данных таблицы и их изменение с точки зрения приложения реализуются максимально просто. Приложение видит табличный файл, просматривает его и вносит изменения в отдельные поля, оставляя в стороне механизм такой работы. С точки зрения реализации объект TTable не представляет собой чего-либо нового. Фактически при работе с ним выдается множество SQL-запросов к базе, как на чтение, так и на запись. Объект только предоставляет пользователю определенный сервис. Это важно при практическом программировании, но несущественно для понимания механизмов доступа к базе из приложений на C++Builder и Delphi.

Использование средств InterBase Express при работе с InterBase

InterBase Express (IBX) представляет собой набор компонентов, которые обеспечивают средства доступа к данным в базах данных InterBase. Этот набор включает

TIBDatabase	TIBTransaction	TIBTable
TIBQuery	TIBDataSet	TIBStoredProc
TIBSQL	TIBUpdateSQL	TIBSQLMonitor
TIBDatabaseInfo	TIBEvents	

При использовании IBX работа с базой данных осуществляется практически напрямую, без использования BDE. IBX позволяет использовать все возможности InterBase и за счет исключения промежуточных звеньев (BDE) повысить скорость обработки данных.

Хотя компоненты IBX в основном аналогичны BDE-компонентам, тем не менее, имеется ряд отличий. Приведем эти отличия для основных компонентов IBX.

Компонент **TIBDatabase** используется, чтобы установить связь с базами данных. В базе данных может одновременно выполняться несколько транзакций. В отличие от BDE, IBX имеет отдельный компонент, который позволяет отделять подключение к базе данных от транзакций. Рассмотрим основные свойства объекта.

DatabaseName Имя базы данных. Для локального подключения - диск, путь и имя файла базы данных.

Connected Указывает, установлена ли связь с базой данных, заданной свойством **DatabaseName** (значение **true** или **false).** Установка свойства **Connected** в **true** эквивалентна выдаче SQL-команды CONNECT, установка свойства **Connected** в **false** эквивалентна выдаче SQL-команды DISCONNECT.

Имя пользователя и пароль можно сохранить в свойстве **Params** компонента **TIBDatabase:**

Пример 10.22

```
User_name=sysdba
Password=masterkey
```

В отличие от Borland Database Engine, IBX управляет транзакциями с помощью отдельного компонента - **TIBTransaction.** Это позволяет отделить транзакции от подключений к базам данных, так что можно реализовать преимущества InterBase при двухфазном завершении транзакций при работе с несколькими базами.

Компонент **TIBTransaction** используется для явного управления транзакциями, обеспечивая возможность, если это необходимо, иметь несколько параллельных транзакций, работающих с одной или несколькими базами. Следует, правда, заметить, что в большинстве случаев вполне достаточно одной транзакции в одной базе. Рассмотрим основные свойства объекта.

DefaultDatabase Задает базу данных, для которой запускается транзакция. В случае работы с несколькими базами список баз данных формируется с использованием метода AddDatabase().

Active Указывает, активна **(Active=true)** или нет **(Active=false)** транзакция. Задание **Active=true** стартует транзакцию.

Params Содержит список параметров транзакции. В качестве параметров задается уровень изоляции, обработка конфликтов.

Например, для уровня изоляции SNAPSHOT можно задать
concurrency
nowait.
Для уровня изоляции READ COMMITTED можно задать
read_committed
rec_version
nowait.

Компонент **TIBQuery** позволяет выполнить любую InterBase-команду DSQL. Рассмотрим основные свойства объекта.

SQL Содержит текст команды на SQL, подлежащей выполнению (аналогично свойству SQL-объекта TQuery).

Params Содержит список параметров запроса. При выполнении запроса значения параметров подставляется в выражение SQL, после чего запрос компилируется. Это позволяет динамически формировать запросы, имитируя параметры, хотя сам SQL их и не поддерживает (аналогично свойству **Params** объекта TQuery).

Prepared Признак готовности запроса к выполнению: **Prepared=true** - запрос подготовлен, **Prepared=false** - нет. Установка **Prepared=true** (или вызов метода Prepare()) вызывает компиляцию запроса; последнее существенно при многократно используемых запросах с параметрами (аналогично свойству **Prepared** объекта TQuery).

Database Идентифицирует базу данных, указывая на компонент TIBDatabase с которой работает запрос. При работе с BDE для ссылки на базу данных используется в большей мере свойство **DatabaseName** (свойство **Database** доступно только для чтения), а в IBX такого свойства просто нет.

Active Указывает, открыт **(Active=true)** или нет **(Active=false)** запрос. Применяется только к запросам, содержащим SQL-команду SELECT. Установка **Active** в **true** или **false** соответственно открывает или закрывает запрос (можно также использовать методы Open() и Close()). Свойство **Active** аналогично свойству **Active** объекта TQuery.

Transaction Идентифицирует транзакцию, под управлением которой выполняется запрос.

При использовании **TIBQuery** для выборки данных результаты выборки доступны, в отличие от **TQuery,** только для чтения. Для коррекции данных, полученных с помощью **TIBQuery,** следует использовать объекты типа **TIBUpdateSQL.**

Компонент **TIBDataSet,** как и **TIBQuery,** позволяет выполнить любую InterBase-команду DSQL. Помимо этого данный объект позволяет корректировать считанные по SELECT данные. Рассмотрим основные свойства объекта.

Вместо свойства **SQL,** описывающего действия с базой в объектах **TIBQuery** и **Tquery,** объекты **TIBDataSet** содержат пять свойств: **SelectSQL, RefreshSQL, DeleteSQL, InsertSQL, ModifySQL.** Они предназначены для хранения соответствующих SQL-команд. Таким образом, Данный объект содержит, практически, полный набор средств для обработки данных.

Приведем пример заполнения этой группы свойств.

Пример 10.23

SelectSQL

```
SELECT BOOKNM,unikey from TBOOK
```

RefreshSQL

```
SELECT BOOKNM,unikey from TBOOK WHERE unikey = : unikey
```

ModifySQL

```
UPDATE TBOOK SET BOOKNM = :BOOKNM
 WHERE unikey = :Old_unikey
```

DeleteSQL

```
DELETE FROM TBOOK WHERE unikey = :Old_unikey
```

InsertSQL

```
INSERT INTO TBOOK (unikey, BOOKNM) VALUES (:unikey,
:BOOKNM)
```

Компонент **TIBEvents** используется для регистрации интереса и обработки событий, зарегистрированных сервером InterBase. Рассмотрим основные свойства объекта.

Database Связывает регистрируемые события с конкретной базой данных. Свойство **Database** указывает на объект **TIBDatabase.**

Events Задает список контролируемых событий. Список может содержать до 15 событий.

Registered Указывает, что были зарегистрированы события **(Registered = true)** или нет **(Registered = false),** перечисленные в свойстве **Events.**

Для регистрации интереса к событиям используется метод **RegisterEvents().** Для получения сведений о произошедшем событии (из перечня, указанного в свойстве **Events)** используется метод QueueEvents(). Предварительно события должны быть зарегистрированы с помощью **RegisterEvents()**.

По существу, объект **TIBEvents** реализует группу команд SQL EVENT INIT, EVENT WAIT.

Компонент **TIBTable** в основном аналогичен BDE-компоненту **TTable.** Компоненты **TIBStoredProc, TIBStoredProc**, **TIBUpdateSQL** также аналогичны соответствующим компонентам **TStoredProc, TstoredProc** и **TUpdateSQL.** С точки зрения анализа работы с InterBase они носят вспомогательный характер.

ОСНОВНЫЕ ЭТАПЫ РАЗРАБОТКИ ПРИЛОЖЕНИЙ

Рассмотрим типичный фрагмент приложения, работающего с базой данных. Фрагмент включает экранную форму с размещенным на ней табличным документом, в котором осуществляется просмотр и редактирование информации, получаемой из базы данных.

Данные для документа выбираются из базы по запросу. Для формулировки и обработки запроса используется объект TQuery. SQL для сохранения изменений (обновления, модификации, удаления) записывается в объекте TUpdateSQL.

Для визуализации данных используется объект TDBGrid. Удобство навигации обеспечивается с помощью объекта TDBNavigator.

Связи между объектами TQuery, TDBGrid, TDBNavigator реализуются с помощью объекта TDataSource.

Внешний вид формы во время проектирования представлен на рис. 10.1. Объекты TQuery, TDataSource, TUpdateSQL являются невизуальными и потому во время выполнения их нет на форме.

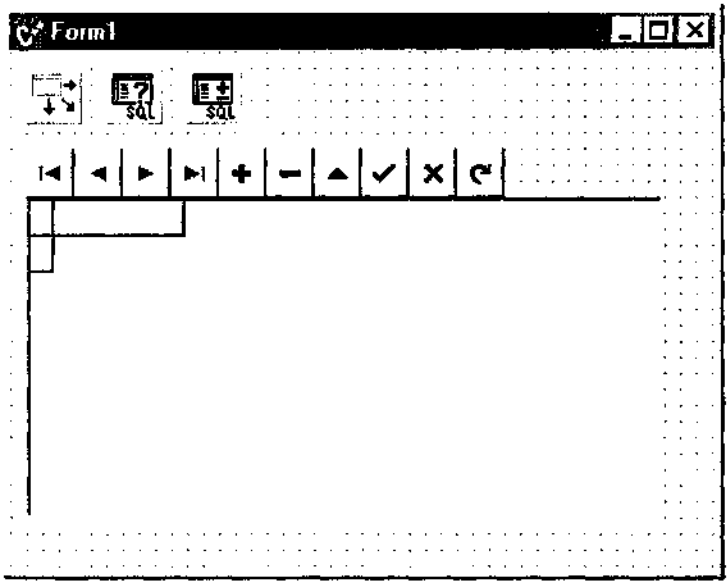


Рис. 10.1. Форма с размещенными на ней компонентами:
- верхний ряд - DataSource1, Query1, UpdateSQL1;
- средняя полоса - DBNavigator1;
- нижнее окно - DBGrid1

По умолчанию размещенным объектам присваиваются имена, включающие имя объекта и его порядковый номер на форме. При желании их

можно задать и явно. В нашем случае в этом нет необходимости, поэтому объекты получат имена *DataSourcel, Query], UpdateSQL1, DBGridl, DBNavigator1.*

Размещение компонентов на форме осуществляется выбором соответствующих объектов из палитры инструментов и помещением их на форму. Размеры и расположение визуальных компонентов осуществляются их перетаскиванием и растяжением «мышью».

Теперь можно перейти к настройке приложения на работу с базой.

Прежде всего, установим связи между объектами. Для указания связи визуальных компонент с источником данных необходимо задать их свойство с **DataSource.** В нашем случае *DataSource1.* Удобнее всего задавать свойства, использую инспектор объектов Object Inspector (рис. 10.2).

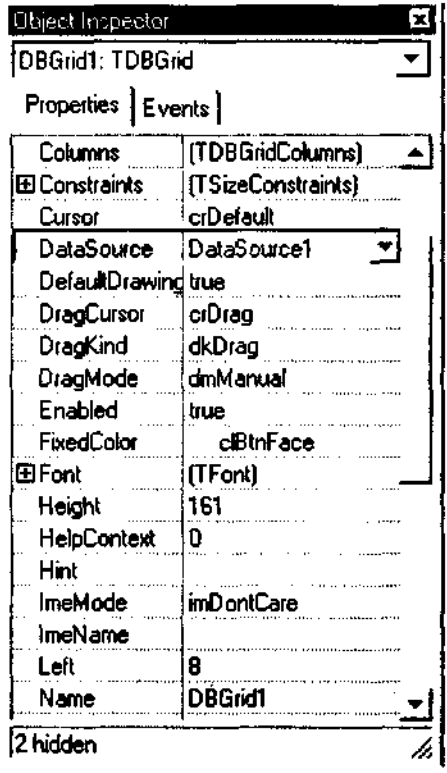


Рис. **10.2.** Инспектор объектов для DBGridl с выбранным полем DataSource.

Установим свойство **DataSource** объектов *DBGridl* **и** *DBNavigatorl* **в** *DataSourcel.*

Теперь свяжем *DataSourcel* с реальными данными, выбираемыми из базы. В нашем случае - *Query 1,* Связь осуществляется заданием свойства **pataSet** объекта *DataSourcel* в *Query 1.*

Поскольку данные предполагается корректировать, то необходимо связать *Query1* с объектом, содержащим SQL для корректировки - *UpdateSQL1*. В TUpdateSQL предусмотрена запись трех команд SQL. Команды могут быть, вообще говоря, любыми. Их запуск осуществляется по инициативе прикладной программы. Стандартно предполагается их вызов для команд удаления - Delete *(DeleteSQL),* вставки - Insert *(InsertSQL),* обновления - Update (*ModifySQL*).

Результаты выборки данных из базы, полученные с помощью объектов TQuery, представляют собой виртуальную таблицу. Эта таблица с точки зрения ее обработки может быть нескольких видов.

- Только для чтения.
- Для прямой корректировки. В этом случае любое изменение в TQuery непосредственно записывается в базу. Такая корректировка, правда, возможна только тогда, когда данные выбираются из одной таблицы и множества строк и столбцов выборки являются подмножествами строк и столбцов исходной таблицы, причем выборка не содержит вычисляемых данных. В этом случае необходимые команды SQL при внесении изменений в TQuery генерируются автоматически средствами BDE. Такой подход в ряде случаев очень удобен, но перечисленные выше ограничения часто бывают слишком обременительными.
- Для корректировки с записью результатов с помощью явно указанных команд SQL. В этом случае никаких ограничений на TQuery нет, но за это приходится платить необходимостью явного описания команд обновления. В Query данные при этом могут копиться, как в локальной таблице, вне прямой связи с данными в базе. Это позволяет проводить запись изменений пакетом, например, при полном завершении корректировки. На необходимость того, что данные должны копиться, указывает свойство **CashedUpdate: CashedUpdate=true** - должны копиться, **CashedUpdate=false** - нет. Помимо этого необходимо указать на объект, содержащий команды обновления - TUpdateSQL. Если хотя бы одно из них не будет указано надлежащим образом, TQuery будет недоступно для внесения изменений.

Рассмотрим подробнее последний вариант, как наиболее мощный и гибкий.

Следует заметить, что помимо перечисленных явно объектов, неявно используется еще один - TDataBase. Данный объект имеет свой набор

свойств и методов, в частности, механизм управления транзакциями реализуется именно через него.

Схема связей перечисленных объектов в рассматриваемом случае представлена на рис. 10.3.

При описании объектов связи реализуются указанием имен связываемых объектов в соответствующих свойствах.

Работа ввода данных в визуальные объекты, в нашем случае - DBGrid1, состоит в обработке возникающих при этом событий. С объектами связано некоторое множество событий. Для любого события из этого множества можно задать программу обработчика. При работе с базами данных наиболее активно используются следующие события:

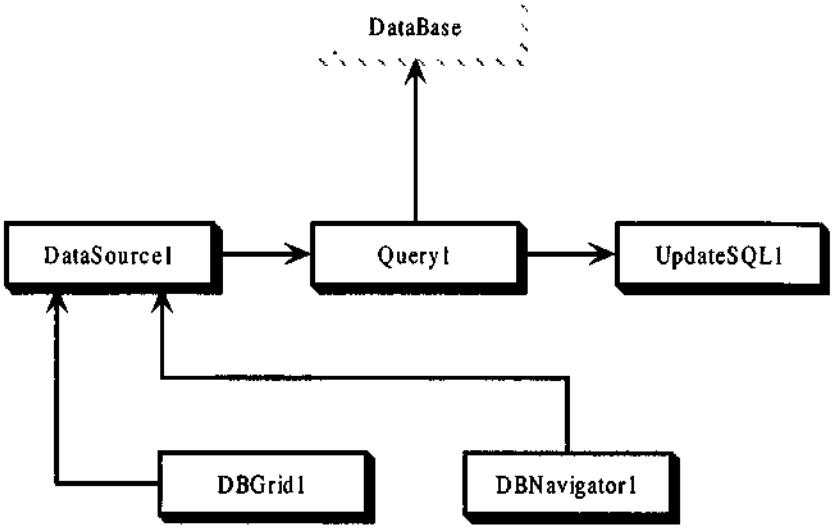


Рис. 10.3. Схема связей объектов.

BeforeDelete	AfterDelete
BeforeInsert	AfterInsert
BeforePost	AfterPost

Эти события возникают при удалении, вставке и фиксации изменений в строках обрабатываемого набора данных, в нашем случае - Query1. Первая группа возникает непосредственно перед внесением изменений, вторая - сразу же после их внесения.

Тот факт, что с внесением изменений связано не одно, а пара событий, позволяет обеспечить адекватную реакцию на них.

Рассмотрим подробнее обработку этих событий и ее особенности от момента возникновения события «до» или «после».

Пример 10.24

```
void__fastcall TForm1::Query1BeforeInsert(TDataSet
*DataSet)
{
 if (. . .)  // Проверка условия допустимости вставки
 {Abort(); // Вставка недопустима
  return;
 }
 . . .      // Вставка допустима, выполняем
// подготовку к вставке данных
}
```

Здесь проверяется условие возможности внесения новых строк. Если внесение изменений в данный момент запрещено, то вызывается функция Abort(), результатом действия которой является отмена вставки. Если действие разрешено, то выполняются необходимые подготовительные действия (возможно и "никакие"). Необходимо при этом помнить, что сама строка еще *не внесена.*

События BeforeInsert, AfterInsert являются результатом попытки добавить строку в набор данных (в нашем случае - Query1). Какие действия пользователя или приложения могут вызвать подобную ситуацию?

- Нажатие в навигаторе (DBNavigator1) кнопки [+].
- Нажатие клавиши [↓], когда курсор находится в последней строке набора данных при его просмотре (в DBGrid1).
- Вызов приложением метода Append объекта класса TDataSet (Query1→ Append();).

Отметим, что первые действия косвенно порождают вызов того же метода Append.

Следующее событие также связано с добавлением новой строки.

Пример 10.25

```
void__fastcall TForm1::Query1AfterInsert(TDataSet
*DataSet)
{
 // Заполнение полей в новой строке
 // FieldByName(". . .") адресует имя поля,
 // заданное параметром, конструкции AsString, . . .
 // описывают тип возвращаемого или присваиваемого
 // значения

 Query1->FieldByName(". . .")->AsString=". . .";
 Query1->FieldByName(". . .")->AsInteger=567;
```

```
 Query1->FieldByName(". . .")->AsFloat=567.54;
 . . .
}
```

Здесь заполняются значения полей внесенной строки. В результате мы получаем не пустую, а уже заполненную строку. Характер заполнения определяется, естественно, нуждами конкретного приложения. При этом мы можем, например, заполнять те поля, которые не разрешаем корректировать пользователю, более того, значения этих полей могут не печататься на экране вообще.

Отметим, что при всех этих действиях изменения фиксируются только в приложении, а не в базе данных. Внесение изменений в базе производится отдельными командами и в то время, когда это будет нужно приложению. Если осуществляется ввод или коррекция документа, то, скорее всего, это будет момент окончания работы с документом, а не его отдельными строками, которые в процессе работы могут изменяться многократно.

По факту фиксации в наборе внесенных изменений возникают события BeforePost (непосредственно перед фиксацией) и AfterPost (сразу же после нее).

Post-события могут быть результатом действий пользователя (точнее, стандартной обработки этих действий) или приложения. Вот эти действия:

- Нажатие в навигаторе (DBNavigatorl) кнопки .
- Переход от измененной строки к другой строке в наборе. Последнее может быть с точки зрения пользователя результатом:
- Щелчок мышью на другой строке в таблице, выводимой в окне DBGrid, или другом визуальном объекте для работы с базами данных.
- Нажатие клавиш управления курсором  ,  , Page Up, Page Down.
- Нажатие в навигаторе (DBNavigatorl) одной из кнопок 

  Любая из них вызывает перемещение по набору данных и, как следствие, требует, чтобы информация в ранее измененной строке была зафиксирована.
- Вызов приложением метода Post объекта класса TDataSet (Query1→ Post();).

При обработке различия между событиями BeforePost и AfterPost обычно невелики. Единственное, что может иметь существенное значение, так это то, что при обработке BeforePost можно дополнительно внести изменения в строку. Внесение таких изменений при обработке AfterPost нежелательно, так как может привести к возникновению собы-

тия Post во время его же обработки и зацикливанию приложения. В некоторых случаях может также оказаться полезной возможность разделения обработки на два этапа, однако, как правило, достаточно одной программы, обрабатывающей событие Post. Примеры подобных программ обработчиков приведены ниже.

Пример 10.26

```
void __fastcall TForm1::Query1BeforePost(TDataSet *DataSet)
{
 // Заполнение полей в измененной строке
 // FieldByName(". . .") адресует имя поля,
 // заданное параметром, конструкции AsString, . . .
 // описывают тип возвращаемого или присваиваемого
 // значения
 Query1->FieldByName(". . .")->AsString=". . .";
 Query1->FieldByName(". . .")->AsInteger=567;
 Query1->FieldByName(". . .")->AsFloat=567.54;

 // Обработка результатов коррекции, например
 // подготовка запросов для загрузки информации
 // в базу данных
 . . .
}
```

Пример 10.27

```
void __fastcall TForm1::Query1AfterPost(TDataSet *DataSet)
{
 // Обработка результатов внесенных изменений
 . . .
 // Обработка результатов коррекции, например
 // подготовка запросов для загрузки информации
 // в базу данных
 . . .
}
```

Рассмотрим подробнее обработку удаления.

Процедура удаления вызывается одним из следующих способов:

- Нажатием в навигаторе (DBNavigatorl) **кнопки** [–].
- Вызовом приложением метода Delete объекта класса TDataSet (Query1→ Delete();).

При вызове удаления с помощью навигатора, если не указывать явно в свойстве навигатора ConfirmDelete значение false, будет выводиться Диалоговое окно с требованием подтверждения удаления, после чего строка будет удалена из набора.

Пример 10.28

Схема обработки удаления:

```
void__fastcall TForm1::Query1BeforeDelete(TDataSet
*DataSet)
{
 if(. . .) // Проверка условия допустимости удаления
 {Abort(); // Удаление недопустимо
  return;
 }
  . . .    // Удаление допустимо, выполняем
// подготовку к удалению данных
}
```

Следует заметить, что процедуры обработки удаления имеют определенную специфику. Если добавляемые или изменяемые данные остаются в обрабатываемом наборе и результаты изменений можно сохранить в основной базе в любой момент времени, то удаляемые исчезают из него, поэтому просто так отложить обработку удаления невозможно. Таким образом, все действия, непосредственно связанные с обработкой удаления, необходимо производить в событии BeforeDelete. На момент обработки события AfterDelete в наборе уже нет удаляемой строки, а значит соответствующий обработчик пригоден только для обработки информации на оставшихся строках набора данных.

Пример 10.29

Обработчик события AfterDelete:

```
void__fastcall TForm1::Query1AfterDelete(TDataSet
*DataSet)
{
  // Расчеты на оставшейся части набора данных.
 . . .
}
```

До сих пор мы рассматривали обработку данных внутри приложения, не затрагивая вопросы внесения изменений в базу данных. Перейдем теперь к разбору схемы обновления данных в базе.

Прежде всего, необходимо определиться с моментами внесения изменений. Здесь, по существу, имеются два подхода:

- Внесение изменений сразу же после ввода данных в каждой строке.
- Внесение изменений после ввода данных в целый документ.

Проиллюстрируем эти две схемы примерами.

ВНЕСЕНИЕ ИЗМЕНЕНИЙ СРАЗУ ПОСЛЕ ВВОДА ДАННЫХ В КАЖДОЙ СТРОКЕ

Обработкавставки

Обработку вставки данных в базу имеет смысл проводить только после их добавления в рабочий корректируемый набор (событие Afterlnsert), поскольку до выполнения самой вставки их просто нет.

Пример 10.30

```
void__fastcall TForm1::Query1AfterInsert(TDataSet
*DataSet)
{
 // Заполнение полей в новой строке
 // FieldByName(". . .") адресует имя поля,
 // заданное параметром, конструкции AsString, . . .
 // описывают тип возвращаемого или присваиваемого
 // значения

 Query1->FieldByName(". . .")->AsString=". . .";
 Query1->FieldByName(". . .")->AsInteger=567;
 Query1->FieldByName(". . .")->AsFloat=567.54;
 . . .

 UddateSQL1->Apply(ukInsert);
}
```

Сразу заметим, что обычно выполнение записи в базу сразу после вставки нецелесообразно. В самом деле, непосредственно после вставки нельзя обеспечить полноту данных. Обычно они еще подлежат корректировке. С другой стороны, набор данных после вставки переходит в режим редактирования (Edit), а это означает, что при любой попытке перехода на другую строку, закрытии набора, не говоря уже о прямой выдаче Post, возникнет пара событий BeforePost, AfterPost. К моменту возникновения последних событий с гораздо большей уверенностью можно говорить о действительном окончании редактирования строки, поэтому запись в базу целесообразно связать именно с этими событиями и исключить из обработчика событий Afterlnsert. При этом правда, необходимо учитывать, при каких условиях сформировалась строка набора: в результате вставки - тогда необходимо выдать команду Insert, или модификации - тогда необходимо выдать команду Update. Для определения того, является ли строка новой, можно использовать хранимые данные, например, если первичный ключ является автоинкрементным, то у «новой» в соответствующем поле будет 0, а у «старой» - ненулевое значение. Кроме того, состояние набора «вставка строки» можно запомнить в рабочей переменной, можно также воспользоваться свойством State редактируемого объекта (в нашем случае Queryl). В последнем случае надо, правда, соблю-

дать некоторую осторожность, чтобы правильно отслеживать траекторию изменений. Пример единой точки обработки занесения данных в базу рассмотрим ниже.

Пример 10.31

```
void__fastcall TForm1::Query1BeforeInsert(TDataSet
*DataSet)
{
 if(. . .)  // Проверка условия допустимости вставки
  {Abort();  // Вставка недопустима
   return;
 }
  . . .      // Вставка допустима, выполняем
// подготовку к вставке данных
}
    //----------------

void__fastcall TForm1::Query1AfterInsert(TDataSet
*DataSet)
{
 // Заполнение полей в новой строке
 // FieldByName(". . .") адресует имя поля,
 // заданное параметром, конструкции AsString, . . .
 // описывают тип возвращаемого или присваиваемого
 // значения

 Query1->FieldByName(". . .")->AsString=". . .";
 Query1->FieldByName(". . .")->AsInteger=567;
 Query1->FieldByName(". . .")->AsFloat=567.54;
 . . .

 Query1->Tag=1; // Выполнена вставка, записи в базу
                //не было
}
    II----------

void__fastcall TForm1::Query1BeforePost(TDataSet *DataSet)
{
 // Заполнение полей в измененной строке
 // FieldByName (". . .") адресует имя поля,
 // заданное параметром, конструкции AsString, . . .
 // описывают тип возвращаемого или присваиваемого
 // значения
 Query1->FieldByName(". . .")->AsString=". . .";
 Query1->FieldByName(". . .")->AsInteger=567;
 Query1->FieldByName(". . .")->AsFloat=567.54;
```

```
 // Обработка результатов вставки, например
 // подготовка запросов для загрузки информации
 // в базу данных
 . . .
}
    //----------------

void__fastcall TForm1: :Query1AfterPost(TDataSet *DataSet)
{
 // Обработка результатов внесенных изменений
 . . .
 // Обработка результатов вставки, например
 // подготовка запросов для загрузки информации
 //в базу данных
  . . .
 UpdateSQL->Apply((Query1->Tag)?ukInsert:ukModify);
 Query1->Tag=0;
}
```

Обработка замены

Собственно замена данных в базе сосредоточивается в обработчиках событий BeforePost, AfterPost. Основная схема показана в приведенном выше примере. Единственное, что стоит добавить, так это то, что в некоторых случаях стоит предварительно проверять, произошли в обрабатываемом наборе реальные изменения или нет.

Обработка удаления

Удаление данных в базе наиболее естественно проводить, как уже отмечалось выше, в обработчике события BeforeDelete.

Пример 10.32

```
void__fastcall TForm1::Query1BeforeDelete(TDataSet
*DataSet)
{
 if(. . .) // Проверка условия допустимости удаления
 {Abort(); // Удаление недопустимо
  return;
 }
 • . .     // Удаление допустимо, выполняем
// подготовку к удалению данных
 UpdateSQL->Apply(Query1->Tag)?ukDelete);
}
```

Отметим, что во всех приведенных выше примерах никак не фигурировал механизм управления транзакциями. При таком подходе каждое обновление порождает свою транзакцию. Поэтому допускается запись

в базу части документа. Кроме того, нигде не указана обработка неудачного выполнения действий по записи информации в базу данных.

Общую схему обработки успешности записи можно проиллюстрировать следующим примером.

Пример 10.33

```
try  {
 TDatabase *DB=Query1->Database;
// лучше запомнить указатель на объект Database,
// описывающий базу данных однократно при ее
// открытии
 DB->StartTransaction();
 UpdateSQL1->Apply(ukModify);
 DB->Commit();
    }catch (const Exception &E){

 int i;
 AnsiString ErrMes=E.Message,
 DiagMes="Строка документа. . .";
// Выделение части сообщения, порождаемого Exception
//на русском языке (коды кириллицы < 0); если такой
// части нет, то сохраняется полный текст

 for(i=1;(i<=ErrMes.Length())&&(ErrMes[i]>'\0');
    i++);
 if(i<=ErrMes.Length())
  ErrMes=ErrMes.SubString(i,ErrMes.Length());
 ShowMessage(ErrMes);
 DB->Rollback();
 Application->MessageBox(DiagMes.c_str(),
   "Предупреждение",16);
}
```

ВНЕСЕНИЕ ИЗМЕНЕНИЙ ПОСЛЕ ВВОДА ДАННЫХ В ЦЕЛЫЙ ДОКУМЕНТ

Если документ необходимо либо записать в базу целиком, либо не записывать вообще, то приведенная выше схема является неудачной. В этом случае следует копить изменения, а затем записывать их одним пакетом.

Проблема, которая при этом возникает: что делать с удаленными строками? Вариантов решения может быть много. Рассмотрим два из них.

Первый вариант предусматривает запоминание первичных ключей удаляемых записей. Для этого создаем список и помещаем в него ключи удаленных строк. Можно воспользоваться объектами для работы со списками, а можно создать свой список. Пусть в нашем наборе первичным

ключом служат поля Fieldl, Field2, Field3. Тогда можно использовать следующую конструкцию.

Пример 10.33

```
static struct MFields
{AnsiString a,b,c;
 MFields *d;
};

static struct {AnsiString a,b,c;} WField;

static struct MList
{MFields *p;

 Add(AnsiString x, AnsiString y, AnsiString z)
 {MFields *q=new MFields;
  q->a=x;
  q->b=y;
  q->c=z;
  q->d=p;
  p=q;
 };

 int Get(void)
 {if(p)
  {MFields *q=p;
   WField.a=p->a;
   WField.b=p->b;
   WField.c=p->c;
   p=p->d;
   delete q;
   return 1;
  }else return 0;
 };

 MList(){p=0;};

 ~MList()
 {MFields *q;
  while(p)
   {q=p;
    p=p->d;
    delete q;
   };
 };
};

MList wl;
```

При начале работы, например при запуске формы (событие OnShow), выдаем

```
wl.p=0;
```

В обработчик **события** BeforeDelete **вписываем команду** вида:

```
wl.Add(Query1->FieldByName("Field1")->AsString,
 Query1->FieldByName("Field2")->AsString,
 Query1->FieldByName("Field3")->AsString);
```

Таким образом, мы создаем список удаленных строк. Перед выдачей команды следует, конечно, убедиться, что строка не была только что создана. В этом случае ее нет в основной базе, и удаление из рабочего набора должно проводиться без всяких последствий.

Другой вариант состоит в том, что в набор, подлежащий коррекции (Query), при обращении к базе записывается дополнительное поле, например 0. При удалении сама строка не удаляется, а в это поле записывается 1, являющаяся признаком удаления. При выдаче на экран строки фильтруются по этому признаку, так что для пользователя удаление выглядит обычным образом.

Пример 10.34

```
void__fastcall TForm1::Query1BeforeDelete(TDataSet
*DataSet)
{
 if(. . .) // Проверка условия допустимости удаления
 { // Удаление допустимо
  Query1->FieldByName("PDELETE")->AsInteger=1;
 }Abort();
}
```

Запись данных **в** базу при подобном подходе реализуется в цикле. В этом случае последовательно просматриваются строки набора данных и в зависимости от характера внесенных изменений выдаются команды на вставку, обновление или удаление. Перед началом цикла стартуется транзакция, в конце выдается Commit.

При работе с объектом UpdateSQL следует помнить, что при стандартном использовании параметры запросов на обновления выбираются из объекта Query, который ссылается на данный UpdateSQL. Этих данных иногда бывает недостаточно. Тогда для записи удобнее использовать отдельный объект Query, параметры SQL которого можно задавать программно. Кроме того, можно непосредственно в программе формировать текст SQL-запроса как символьную строку, помещать ее в свойство SQL→ Text объекта Query и выдавать команду на исполнение.

Пример 10.35

```
AnsiString s, s1, s2;
. . . .
s="UPDATE "+s1+s2;
Query1->SQL->Text=s;
Query1->ExecSQL();
```

ОДНОВРЕМЕННАЯ РАБОТА С НЕСКОЛЬКИМИ НАБОРАМИ ДАННЫХ

Все, что рассматривалось выше, относилось к случаю, когда работа ведется с одной выборкой данных. Этого во многих случаях недостаточно. Упомянем о некоторых средствах для связывания нескольких наборов.

Иерархически связанные наборы

При работе часто приходится иметь дело с данными, организованными иерархически, когда каждый следующий уровень детализирует предыдущий.

Пусть набор B детализирует данные набора A. Рассмотрим пример.

Для редактирования на экран выводятся две табличные формы DBGridl и DBGrid2 с соответствующими навигаторами DBNavigatorl и DBNavigator2. DBGridl связан с Query1 (точнее, связи DBGridl → DataSourcel → Queryl и DBNavigatorl → DataSourcel → Queryl). DBGrid2 связан с Query2 (точнее, связи DBGrid2 → DataSource2 → Query2 и DBNavigator2 → DataSource2 → Query2).

Queryl соответствует набору A. Query2 соответствует набору B. Наборы данных связаны по значениям полей: поле F1 набора A соответствует полю G1 набора B, поле F2 набора A соответствует полю G2 набора B.

Внешний вид формы представлен на рис. 10.4.

Для указания связи наборов в свойстве SQL объекта Query2 в конструкции WHERE команды SELECT записывается:

WHERE G1=:F1 AND G2=:F2

В обычном понимании этого конечно недостаточно, поскольку при перемещении по первому набору поля связи будут меняться.

Для автоматизации решения проблемы автоматической перестройки второго набора в свойстве набора Query2 DataSource указывается DataSourcel. В этом случае отслеживание связи осуществляется стандартными средствами, которые выполняют модификацию запроса в зависимости от значения соответствующих полей в Queryl. Прикладной программе делать больше ничего не надо. Схема внесения данных при этом не меняется.

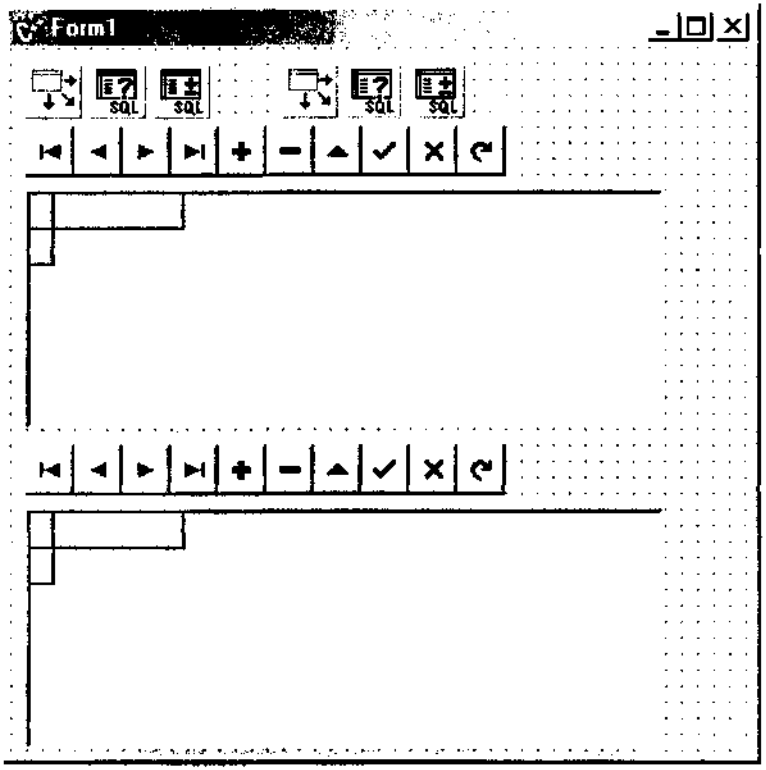


Рис. 10.4. Форма с размещенными на ней компонентами для пары связанных наборов.

Подключение справочной информации

В ряде случаев желательно связать с данными одного набора дополнительную информацию из другого.

Одним из путей является подключение так называемых LookUp полей. Суть их состоит в том, что со строками одного набора связываются поля из другого набора. Для этого в редакторе полей первого набора добавляется "новое" как поле LookUp. Далее указывается, из какого набора выбирается это поле и условия связи в виде перечня ключевых полей. По сути, это эквивалентно добавлению в SQL поля, реализуемого подзапросом.

Теперь с данным полем можно работать так же, как и со всеми остальными, за естественным исключением запрета на его непосредственную корректировку. То есть поддержка навигации по набору осуществляется целиком стандартными средствами, ничего не требуя от прикладной программы.

Приведенные выше примеры, конечно, не исчерпывают всех аспектов работы с базой данных (это предмет отдельной книги). В них рассматриваются лишь отдельные моменты, наиболее часто встречающиеся в приложениях, и те преимущества, которые дает использование систем C++Builder и Delphi при работе с базами данных.

При непосредственном использовании объектов, ориентированных на InterBase (TIBDatabase, TIBTransaction, TIBTable, TIBQuery, TIBDataSet, TIBStoredProc, TIBSQL, TIBUpdateSQL), логика прикладной программы практически не меняется. Различия состоят только в несколько больших возможностях по обработке транзакций и некоторых нюансах в составе свойств объектов и, поскольку они прямо ориентированы на InterBase, естественно, несколько большей скорости выполнения. В силу аналогичности этих объектов отдельно рассматривать работу с ними представляется нецелесообразным.

Глава 11

Инструментальные средства для работы с InterBase

Процесс создания любой информационной или информационно-управляющей системы начинается с проектирования ее информационной базы. Результатом такого проектирования является формализованная логическая структура базы. После этого наступает процесс собственно создания базы данных.

Перед наполнением базы данных содержательной информацией необходимо определить и записать ее структуру, то есть необходимо выполнить ряд действий по созданию метаданных и их записи. Рассмотрим последовательность действий по созданию базы при условии, что ее логическое проектирование уже выполнено:

- Физически создать базу и описать ее общие свойства. Результатом этих действий является, как минимум, файл базы данных xxx.gdb. Создание осуществляется командой CREATE DATABASE.
- Создать комплекс доменов, задающих перечень типов данных, которые будут храниться и обрабатываться в базе. При создании доменов необходимо, насколько это возможно, предусмотреть методы контроля хранимых данных.
- Создать содержательное описание доменов с указанием, где это возможно, предполагаемой области их применения.
- Создать систему таблиц, представляющих собой основные логические блоки, из которых строится база данных.
- Создать содержательное описание назначения таблиц и их отдельных столбцов. Если какие-то столбцы играют роль признаков, то указать возможные значения признаков и их смысл.
- Создать систему ограничений логической целостности базы данных, включающую:
  - создание первичных ключей таблиц;
  - создание внешних ключей таблиц;

- создание CHECK ограничений на группы столбцов таблиц.

- Подготовить сообщения, выдаваемые при возникающих ошибках (EXCEPTION).
- Внести описания условий, при наступлении которых должны выдаваться эти сообщения.
- Подготовить генераторы, если предполагается использование автоинкрементных первичных ключей.
- Создать триггеры для таблиц.
- Создать содержательное описание триггеров таблиц с комментариями, если это необходимо, в тексте самих триггеров. Особое внимание стоит уделить триггерам, работающим с несколькими таблицами.
- Определить и описать вычисляемые поля в таблицах, особенно, если при вычислении используются данные других таблиц.
- Создать и включить в базу хранимые процедуры.
- Создать содержательное описание хранимых процедур с комментариями, если это необходимо, в тексте процедур.
- Подготовить библиотеки пользовательских программ.
- Внести описания пользовательских программ в базу и создать описания назначения пользовательских программ и их интерфейса.
- Определить состав пользователей системы.
- Определить права доступа пользователей к объектам базы данных.
- Определить права доступа хранимых процедур и триггеров к объектам базы данных.
- Подготовить и внести в базу описания обзоров и других дополнительных элементов.

Приведенный перечень работ может показаться излишне подробным и не имеющим прямого отношения к теме главы. Тем не менее, именно из этого перечня виден тот круг задач, которые должны решаться проектировщиком базы данных. Соответственно, если мы говорим об инструментальных средствах для работы с InterBase, то они должны как раз обеспечить возможность решения этих задач с максимальной полнотой и минимальными затратами времени разработчика. К тому же неплохо бы при этом иметь и средства для формирования отчетной документации по базе данных.

Стоит сразу заметить, что все перечисленные задачи, кроме, пожалуй, печати документации, можно решить средствами любой программы, которая умеет выдавать команды на SQL. Вопрос только в том, насколько быстро и удобно будет идти процесс разработки.

В настоящее время известно довольно много средств для ведения работ такого рода - от примитивных утилит до мощных пакетов, таких как ERWin, обеспечивающих работу с самыми разнообразными базами данных и на различных платформах. Мы остановимся только на работе в среде Windows и только применительно к двум из них.

Во-первых, это утилита WinSQL или IBConsole (для версий, начиная с 6), входящие в состав поставки InterBase. Одна из них обязательно есть у каждого пользователя.

Во-вторых, это пакет QuickDesk, отличающийся весьма малым размером, прекрасным интерфейсом, удобным представлением данных, наличием средств отладки и документирования. В общем, я выбрал этот пакет потому, что, по моему, QuickDesk в настоящее время является лучшим пакетом для работы с InterBase, во всяком случае, в своей "весовой" категории.

11.1. WinSQL

Windows ISQL (WISQL) предназначен для интерактивного ввода команды SQL в InterBase.

WISQL запускается либо из меню Windows, либо из командной строки - wisql32.exe.

Окно WISQL после запуска показано на рис. 11.1.

Если Вы работаете с IBConsole, то после соединения с сервером необходимо выбрать пункты меню Tools, Interactive SQL, после чего попадете в окно Interactive SQL. Вид его показан на рисунке 11.2. Функции Interactive SQL в составе IBConsole в основном аналогичны функциям WISQL, поэтому здесь ограничимся только описанием утилиты WISQL.

Система меню WISQL

Главное меню WISQL содержит пункты File, Edit, Session, Query, Metadata и Help. Верхняя половина окна предназначена для ввода команд SQL, нижняя - для вывода полученных результатов.

Меню File содержит команды, предназначенные для выполнения задач создания и удаления базы данных, подключения и отсоединения от базы, запуска SQL-команд из Script-файлов, сохранения результатов вывода и данных сеанса работы в файл, завершение и откат транзакций, а также завершения сеанса работы с WISQL.

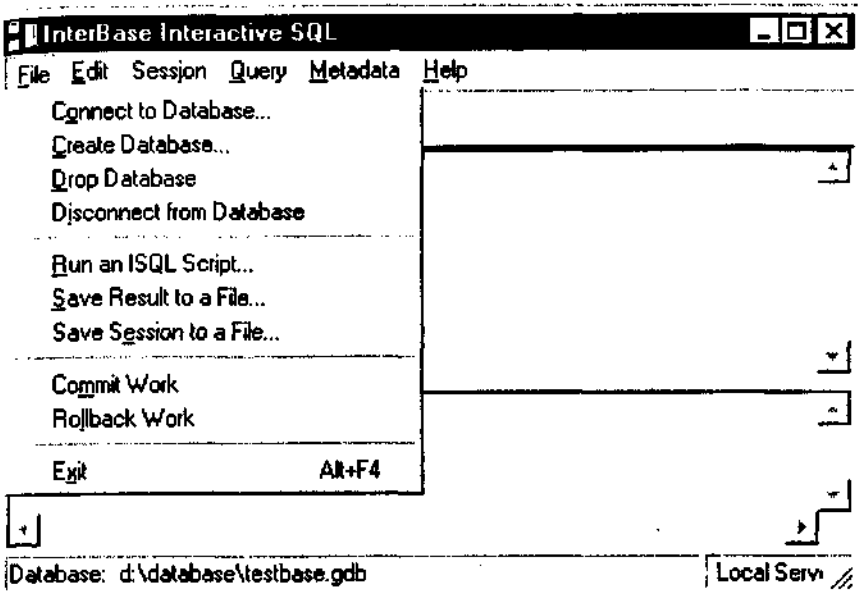


Рис.11.1. Окно и меню Edit утилиты WISQL.

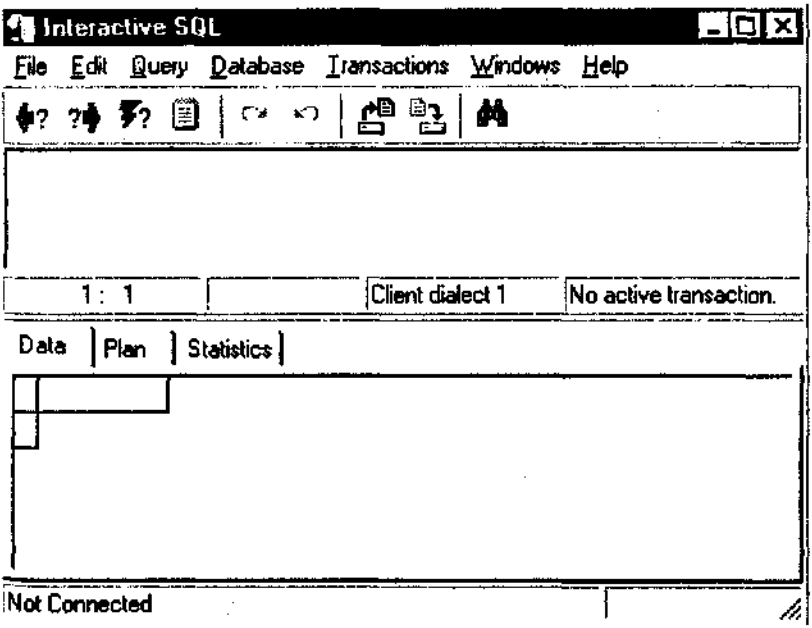


Рис.11.2. Окно Interactive SQL утилиты IBConsole

Меню Edit содержит стандартные команды редактирования данных, включающие копирование и вставку из буфера обмена Windows, отмену последних изменений, выделение набранного текста, очистку окна вывода.

Меню Session содержит команды установки параметров сессии работы с базой данных и выдачи сообщений о текущих установках и версии сервера.

Меню Query содержит команды просмотра, запуска на выполнение набранных SQL команд.

Меню Metadata содержит команды просмотра сведений об объектах базы данных и выдачи описаний таблиц, обзоров и базы данных в целом в виде SQL-скрипта (списка команд создания соответствующих объектов).

Меню Help содержит пункты для доступа к справочной системе по WISQL, а также дает возможность получить некоторые справочные данные по InterBase.

Состав меню и их пунктов может несколько варьироваться в зависимости от используемой версии WISQL.

Кроме того, наиболее активно используемые пункты меню продублированы кнопками.

Замечание. В Help вместо комбинации клавиш Ctrl+ ошибочно указано Alt+.

Порядокработыс WISQL

НАЧАЛО РАБОТЫ С WISQL

Сразу после старта WISQL необходимо соединиться с интересующей вас базой данных или, если предполагается работа с новой базой, создать ее.

Создание базы данных выполняется запуском пункта Create Database меню File. После этого появляется диалоговое окно с полями, описывающими сервер, путь доступа к базе, имя пользователя и пароль. Если сервер запущен на вашей машине, то будет доступен режим Local. В этом режиме необходимо указать полный путь и имя базы с расширением GDB в окне Database.

В удаленном режиме необходимо указать в окне Server имя доступного сервера и выбрать доступный протокол связи из списка в окне Network Protocol. После этого, аналогично предыдущему случаю, указать полный путь, включая путь доступа к серверу, и имя базы с расширением GDB в окне Database.

Далее необходимо заполнить поля User Name и Password.

Кроме того, для указания специфических режимов можно, но необязательно, задать режимы создания базы данных в окне Database **Options.**

Содержание всех заполняемых окон определяется синтаксисом команды ***Create Database,*** так что рассматривать их здесь не имеет смысла.

Если все поля заполнены правильно, то создается база данных (пустая) и выполняется соединение с ней.

Для выполнения соединения необходимо выполнить пункт Connect to Database меню File или воспользоваться кнопкой Connect.

В этом случае высвечивается окно, аналогичное окну создания базы, но без поля Database Options. При наборе имени базы можно воспользоваться выпадающим списком. Далее необходимо заполнить поля User Name, Password и, при необходимости, Role. Содержание всех заполняемых окон определяется синтаксисом команды *Create Connect.*

Заметим, что выполнить эти действия другим способом нельзя. Пока соединение с базой не установлено, доступ к вводу команд SQL закрыт.

ВВОД КОМАНД SQL

После соединения можно переходить непосредственно к работе с базой данных. Эта работа состоит, по существу, в последовательном вводе команд SQL. Команды вводятся в верхнем из двух окон WISQL.

После набора текста команды следует либо вызвать пункт Execute меню Query, либо, что удобнее, воспользоваться кнопкой Execute, либо нажать Ctrl+R.

Результат работы будет показан в нижнем окне. В случае ошибок будет выведено соответствующее диагностическое окно. Полученный результат можно сохранить в файле, воспользовавшись кнопкой Save resalts (Ctrl+S) или пунктом Save Result to a File меню File.

При работе доступны SQL-команды для выборки данных (Select), создания (Insert), модификации (Update) и удаления (Delete). Кроме того, доступны команды описания данных, собственно ради них и нужен WISQL. Это все команды групп Create (для таблиц, доменов, обзоров и т.д.), Alter и Drop.

Доступны также команды для работы с триггерами, хранимыми процедурами и исключениями. Короче говоря, в WISQL доступны все команды ISQL (интерактивного SQL).

Для физического сохранения данных следует набрать команду Commit. Для отказа от изменений - команду Rollback.

При наборе команд можно использовать стандартные средства Windows для копирования данных через ClipBoard.

Кроме того, доступна история выдававшихся команд. Для этого можно использовать кнопки Previous query, Next query или соответствующие пункты меню (горячие клавиши). Высветившийся текст можно откорректировать и вновь отправить на выполнение.

В нижнем окне, если его не очищать, хранятся результаты работы с начала сеанса. Для очистки окна следует перевести курсор в нижнее окно и выполнить пункт Clear Output меню Edit.

В окне задания команд можно указать только одну команду. Для выполнения пакета команд их необходимо поместить в скрипт-файл, после чего пакет можно выполнить, воспользовавшись пунктом Run an ISQL Script меню File.

ЗАВЕРШЕНИЕ РАБОТЫ С WISQL

Для завершения работы с WISQL следует воспользоваться пунктом Exit (Alt+F4) меню File. Перед завершением работы WISQL при необходимости запрашивает, нужно ли сохранять внесенные после последнего Commit изменения.

Для начала работы с другой базой надо выдать команду на разрыв соединения с текущей базой, используя пункт Disconnect from Database меню File или клавишу Disconnect. После этого можно соединиться с новой базой, используя пункт Connect to Database меню File или клавишу Connect.

Дополнительные возможности WISQL

Помимо собственно ввода команд SQL WISQL дает возможность выполнить ряд функций по работе с метаданными, а также с файлами, содержащими набор команд SQL. Кроме того, в рамках WISQL можно управлять форматом вывода данных.

ВЫВОД МЕТАДАННЫХ

Прежде всего следует сказать, для чего нужен вывод метаданных. Во-первых, сведения о метаданных необходимы как справочные данные, без знания которых разработка приложений просто невозможна. Во-вторых, SQL-скрипт необходим, если разработка приложений ведется в одном месте, а эксплуатация приложений - в другом. В этом случае структурные изменения в базе сначала производятся и тестируются в базе разработчика и только затем переносятся на работающую систему. Практически единственный путь - это перенос соответствующих SQL-скриптов.

Вывод метаданных осуществляется на основе пунктов Extract Database, Extract Table и Extract View меню Metadata. В результате выполнения пункта Extract Database формируется полный перечень SQL-команд создания всех объектов базы данных. В результате выполнения пункта Extract Table формируется полный перечень SQL-команд создания указанной таблицы. В результате выполнения пункта Extract View формируется полный перечень SQL-команд создания указанного обзора. Вывод может осуществляться как на экран, так и в файл.

НАСТРОЙКА РЕЖИМОВ РАБОТЫ WISQL

Для настройки режимов работы WISQL служат два диалоговых окна. Доступ к ним осуществляется через пункты Basic Settings и Advanced Settiings меню Session.

Диалоговое окно **Basic ISQL Set Options** дает возможность изменять все основные параметры настройки WISQL, которые могут быть или включены, или выключены. Если установка включена, то она помечается значком "✓". Переключение осуществляется "мышиным кликом" в соответствующем поле.

Рассмотрим перечень настраиваемых параметров:

- **Display Query Plan** (выдача плана выполнения запроса). Если включен, WISQL отображает план запроса, выбранный оптимизатором. Для изменения плана оптимизатора следует явно задавать план, используя опцию PLAN команды SELECT.
- **Auto Commit DDL** (автоматическое сохранение для команд DDL). Если включен (значение по умолчанию), операторы описания набора данных сохраняются в базе (commit) немедленно. Если выключен, инструкции DDL должны быть сохранены (commit) или отменены (rollback) вручную.
- **Display Statistics** (вывод статистики). Если включен, отображает статистику выполнения работ для каждой введенной команды. Ведется следующая статистика:
  - Количество запросов на чтение или запись (Reads = . . .; Writes =)
  - Количество запросов данных или модификаций, которые могут обрабатываться в кэше (Fetches =...).
  - Затраченное время (Elapsed time= ... sec).
  - Использование памяти (Current memory = . . .; Delta memory = . . .; Max memory = ...).
  - Количество используемых буферов базы данных (Buffers = . . .).
- **Display in List Format** (вывод в формате списка). Если включен, WISQL отображает данные в вертикальном формате столбца. Строка состоит из наименования (заголовка) столбца слева и значения справа. Если выключен, WISQL отображает данные в табличном формате.
- **Display Row Count** (вывод количества строк). Если включен, WISQL будет отображать число строк, возвращенных каждым SELECT запросом.

- **Display Time Data Type** (вывод данных типа время). Если включен, время, как и дата, будет отображаться для столбцов типа данных DATE. Если выключен, то только часть даты данных типа DATE будет отображена.

Диалоговое окно **Advanced ISQL Set Options** дает возможность изменять параметры настройки WISQL для работы с BLOB (большой двоичный объект). Если установка включена, то она помечается значком "•". Переключение осуществляется "мышиным кликом" в соответствующем поле.

Опции диалога определяют, как осуществляется вывод BLOB. Команда SELECT всегда выдает идентификатор BLOB для соответствующих столбцов. По умолчанию, SELECT также отобразит фактические данные BLOB для текстовых подтипов.

Можно задать следующие режимы вывода:

- **Disable BLOB Display** (отменить вывод BLOB). WISQL не отображает содержание столбцов BLOB.
- **Display ALL BLOBs** (выводить все BLOB). Выводит BLOB всех типов.
- **Restrict BLOB Display** (ограничить вывод BLOB). Выводит содержание BLOB только для указанного подтипа. Для неизвестного подтипа используется 0; для текстового подтипа (значение по умолчанию) - 1.
- **Character set** (набор символов). Эта установка определяет активный набор символов для строковых данных при всех последующих подключениях баз данных. Это дает возможность отменить заданный по умолчанию набор символов для базы данных. Выбор осуществляется из выпадающего списка доступных наборов символов. Специфицировать набор символов следует до соединения с базой данных.

РАБОТА СО СКРИПТ-ФАЙЛАМИ

WISQL позволяет запуск скрипт-файлов, обеспечивая ввод последовательности заранее подготовленных команд SQL.

Запуск таких файлов на выполнение осуществляется пунктом Run an ISQL Script меню File. Перед выполнением команды выдается вопрос о записи (commit) или отмене (rollback) введенных ранее изменений.

Прежде чем выполнить скрипт-файл, его необходимо предварительно создать. Скрипт-файл готовится, как обычный текстовый файл.

Каждый скрипт-файл ISQL должен начинаться или с команды CREATE DATABASE, или с команды CONNECT (включая имя пользователя и пароль), задавая базу данных, с которой работает скрипт-файл.

Для локальной базы это может выглядеть, например, так:

```
CONNECT "C:\IBLOCAL\EXAMPLES\MYDB.GDB"
USER "myusername" PASSWORD "mypassword";
```

Для удаленной базы задание соединения будет зависеть от способа соединения. Для TCP/IP следует отделить сервер и имя базы данных двоеточием, например:

```
CONNECT "SVR1:VOL2:\USERS\EXAMPLES\MYDB.GDB"
USER "myusername" PASSWORD "mypassword";
```

Для NetWare SPX/TPX следует отделить сервер и имя базы данных символом "@", например:

```
CONNECT "SVR1@VOL2:\USERS\EXAMPLES\MYDB.GDB"
USER "myusername" PASSWORD "mypassword";
```

Чтобы подключиться к серверу Windows NT, используя NetBEUI/Named Pipes, перед именем сервера указывается или двойная наклонная (//), или обратная двойная наклонная (\\), диск, каталог и имя файла отделяются прямыми или обратными наклонными чертами. Например:

```
CONNECT "\\SVR3\D:\USERS\EXAMPLES\MYDB.GDB
"USER "myusername" PASSWORD "mypassword";
```

Скрипт ISQL может содержать любые команды ISQL. Каждый скрипт-файл должен заканчиваться или EXIT, чтобы сохранить (commit) изменения, или QUIT, чтобы отменить (rollback) изменения, сделанные скриптом.

Каждая команда SQL должна быть закончена точкой с запятой (;) либо другим признаком конца, если он был установлен командой SET TERM.

Скрипт может включать комментарии, аналогично программам C. Начало комментария помечается символами "/*", конец - "*/". Комментарий может быть как однострочным, так и многострочным:

```
/* Комментарий */
```

КОМАНДЫ SET, ИСПОЛЬЗУЕМЫЕ В WISQL

Команды SET используются для конфигурирования Windows ISQL непосредственно из скрипт-файла. При использовании Windows ISQL

в интерактивном режиме можно выполнять те же самые функции из меню Session. Сами команды SET не могут задаваться в области ввода команд SQL. Перечень команд SET приведен в табл. 11.1.

Таблица 11.1. *Команды SET конфигурирования Windows ISQL*

Команда	*Описание*
SET AUTODDL	Переключает порядок завершения команд DDL
SET BLOBDISPLAY n	Включает отображение BLOB типа n
SET COUNT	Переключает признак необходимости вывода количества выбранных SELECT строк
SET ECHO	Переключает вывод каждой команды в состояние включено или выключено
SET LIST string	Переключает вывод столбцов из вертикального формата (список) в горизонтальный (таблица)
SET NAMES	Задает используемый символьный набор
SET PLAN	Задает необходимость вывода плана выполнения запроса, созданного оптимизатором
SET STATS	Задает режим вывода (отказа) статистики
SET TERM string	Задает признак конца команды
SET TIME	Задает режим вывода времени для данных типа DATE

По умолчанию, все установки, кроме AUTODDL, отключены, признак конца - ";". При любом запуске сессии WISQL установки получают значения по умолчанию.

После выполнения скрипт-файла все установки приводятся к состоянию до его запуска.

11.2. EMS QuickDesk

Назначение и основные возможности EMS QuickDesk

QuickDesk - продукт, разработанный в России. Владельцем продукта является компания EMS (Electronic Microsystems, Электронные Микросистемы). Авторы: С. Востриков, В. Винокур, Голдобин, Н. Чанов. Под-

робности можно найти через Interrnet по адресу www.ems-hitech.com/quickdesk.

НАЗНАЧЕНИЕ QUICKDESK

QuickDesk предназначен для разработки и управления базами данных InterBase. В основу продукта положен принцип визуального проектирования базы, то есть фактически QuickDesk является интегрированной средой разработчика баз данных InterBase.

QuickDesk содержит специализированные средства документирования базы данных, что дает разработчику возможность создавать текстовые описания практически для всех объектов базы. Описания сохраняются внутри базы в системных таблицах, предусмотренных архитектурой InterBase. Никаких дополнительных таблиц QuickDesk для этого не создает.

Начиная с версии 1.6, QuickDesk обеспечивает подготовку разнообразных печатных отчетов о проектируемой базе. В рамках QuickDesk также реализованы все необходимые средства для администрирования базой данных.

ОСНОВНЫЕ ВОЗМОЖНОСТИ

Создание и удаление базы данных

Создание базы (Create Database) осуществляется в диалоговом режиме, позволяя естественным образом сформулировать требования к создаваемой базе. Удаление (Drop Database) вообще не требует специальных знаний, достаточно только подтверждения соответствующих прав. И та, и другая функция доступны из основного меню Database. Необходимо помнить, что Interbase поддерживает команду DROP DATABASE только для активной базы данных, поэтому необходимо подключиться к базе. В любом другом случае пункт Drop Database является недоступным.

Просмотр элементов базы данных (проводник)

QuickDesk поддерживает одновременную работу с несколькими базами данных. Необходимо зарегистрировать базы в DB Explorer. Для этого можно воспользоваться:

- пунктом основного меню Database→ Register Database;
- пунктом контекстного меню в DB Explorer (Register Database);
- механизмом Drag & Drop, «перетащив» название файла базы данных из проводника Windows (только в версии 2.00).

Формального ограничения на количество зарегистрированных баз данных в QuickDesk не существует.

После подключения к зарегистрированной базе DB Explorer строит дерево объектов, которые содержатся в базе данных. Выделяется девять типов стандартных объектов: домены, таблицы, представления, хранимые процедуры, триггеры, генераторы, исключения, пользовательские функции и роли. Роли поддерживаются только для InterBase начиная с версии 5.0. Двойное нажатие на названии объекта открывает редактор объекта. Кроме того, в зависимости от текущего выбранного элемента дерева становятся доступны различны функции, которые можно вызвать из контекстного меню DB Explorer, появляющегося при нажатии на правую кнопку мыши.

Редактирование SQL запросов

QuickDesk содержит специализированный инструмент для непосредственной работы с SQL-выражениями - SQl Editor. В этот инструмент входит полностью настраиваемый гипертекстовый текстовый редактор, позволяющий выполнять отдельные запросы к базе данных. Если запрос возвращает в результате некоторый набор данных, то этот набор выводится в таблице. Кроме того, SQL Editor содержит страницу с диаграммой анализа производительности запросов. После каждого запроса QuickDesk запрашивает у InterBase системную статистику, позволяющую увидеть, какие именно таблицы участвовали в запросе (включая системные), и были ли использованы при выполнения запроса индексы. Работа с диаграммой позволяет оптимизировать запросы к базе данных.

Редактирование отдельных типов объектов

Для подготовки и редактирования отдельных типов объектов предусмотрены соответствующие средства:

- редактор таблиц / обзоров;
- редактор доменов;
- редактор хранимых процедур;
- редактор триггеров;
- редактор генераторов;
- редактор исключений;
- редактор UDF.

Редактор таблиц позволяет создавать или модифицировать таблицы и данные в таблицах в интерактивном режиме. Создание и модификация генераторов и исключений также максимально упрощена. Что касается хранимых процедур, триггеров, описаний UDF, то это чисто программные объекты. Здесь возможности автоматизации существенно меньше, но использование редакторов помогает избежать синтаксических ошибок и упрощает отладку.

Подготовка выполнения SQL Script

Предусмотрен модуль для редактирования и выполнения скриптов (последовательных наборов sql-выражений). Подобные скрипты могут содержать команды подключения к базе данных или обращаться к текущей активной базе данных в QuickDesk.

Поиск и извлечение метаданных

Обеспечивается поиск строк в текстах процедур, триггеров и обзоров в указанной базе данных. Результат выводится в виде списка найденных объектов. Двойной щелчок на объекте откроет его в редакторе объекта.

Создание скриптов

Все действия над объектами базы данных (имеются в виду изменения структуры базы) можно автоматически сохранять в лог-файл. Полученный скрипт можно выполнять в SQL Script для того, чтобы, например, синхронизировать сделанные изменения в рабочей базе данных.

Можно также создать скрипт при помощи специализированного инструмента - Metadata Extract. После того как вы укажете необходимые объекты базы данных,' QuickDesk создаст скрипт, состоящий из sql-команд, создающих эти объекты.

Печать метаданных

Обеспечивается возможность печати метаданных по всей базе данных или по любой выборке объектов базы данных, позволяющая оперативно создавать документацию по структуре базы данных.

Управление пользователями баз данных и распределением прав доступа к объектам базы данных

Предусмотрен визуальный редактор для управления пользователями базы данных, ролями и редактор для управления распределением прав пользователей и объектов базы данных.

ГЛАВНОЕ МЕНЮ QUICKDESK

При запуске QuickDesk на экран выводится окно, показанное на рис. 11.2 (его вид может несколько отличаться в зависимости от используемой Версии). Здесь приводятся данные по версии 1.7.10.6.

Рис. 11.2. Панель меню и инструментов QuickDesk.

Каждый из пунктов главного меню является заголовком соответствующих подменю.

МЕНЮ DATABASE

Меню Database содержит следующие пункты:

Register Database Перед началом работы с новой базой ее следует зарегистрировать. Для этого необходимо в диалоге выбрать требуемую базу данных, указать нужные данные, включая имя пользователя и пароль. Результаты запоминаются и в дальнейшем используются при соединении с базой.

Create Database При создании базы необходимо задать ее местоположение и ответить на вопросы о требуемых характеристиках базы. В результате создается пустая база со специфицированным именем и характеристиками.

Unregister Database Снятие регистрации очищает в QuickDesk сведения о базе. Сама база при этом не меняется.

Drop Database Физически удаляет базу данных.

Exit Завершает работу QuickDesk.

МЕНЮ VIEW

Меню View содержит следующие пункты:

DB Inspector Активирует окно инспектора объектов базы данных. В инспекторе обеспечивается просмотр перечня объектов базы и их свойств. Инспектор объектов является одним из наиболее удобных средств для слежения за состоянием базы. Подробнее работу с DB Inspector рассмотрим чуть ниже.

Toolbars Управляет составом кнопок, выводимых на инструментальную панель, обеспечивая ее настройку под вкусы пользователя. Функционального назначения не имеет.

Previous Window, Next Window При работе с QuickDesk пользователь может работать с несколькими окнами. Пункты меню Previous Window, Next Window позволяют переключаться между открытыми редакторами.

ИНСПЕКТОР ОБЪЕКТОВ (DB INSPECTOR)

Теперь подробнее рассмотрим работу с инспектором объектов. После своего старта, а обычно он запускается сразу же после запуска QuickDesk, инспектор объектов имеет вид, показанный на рис. 11.3.

В верхней половине окна выведен иерархический список объектов базы данных. Если около объекта стоит (+), то он имеет внутреннюю

структуру, которая может быть развернута "щелчком мыши". В приведенном примере развернут объект **Tables.** В окне показан перечень таблиц базы. Выделена таблица TBOOK_PLACE.

Нижняя половина окна содержит описание свойств выделенного объекта. В данном случае - перечень полей (столбцов) таблицы.

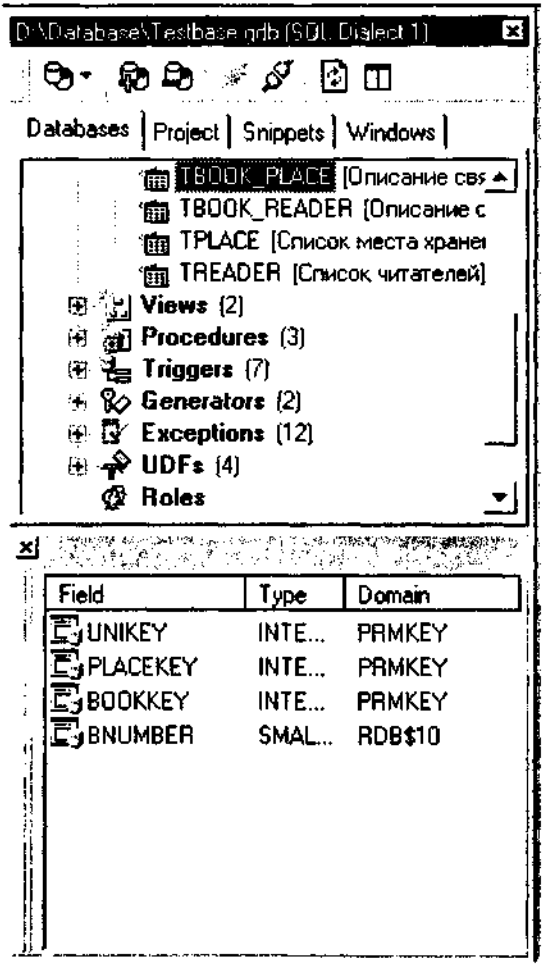


Рис. 11.3. Панель инспектора объектов QuickDesk.

Инструментальная панель вверху DB Inspector дублирует функции, вызываемые из меню. Содержание отдельных кнопок можно посмотреть по всплывающим подсказкам.

Двойным щелчком левой кнопки мыши открывается окно редактиро. вания выделенного объекта.

Щелчком правой кнопки мыши открывается меню, позволяющее создать новый объект выбранного типа, редактировать объект, удалить его, а также выполнить ряд дополнительных функций. Внешний вид этого окна (щелчок был выполнен на объекте-домене) показан на рис. 11.4.

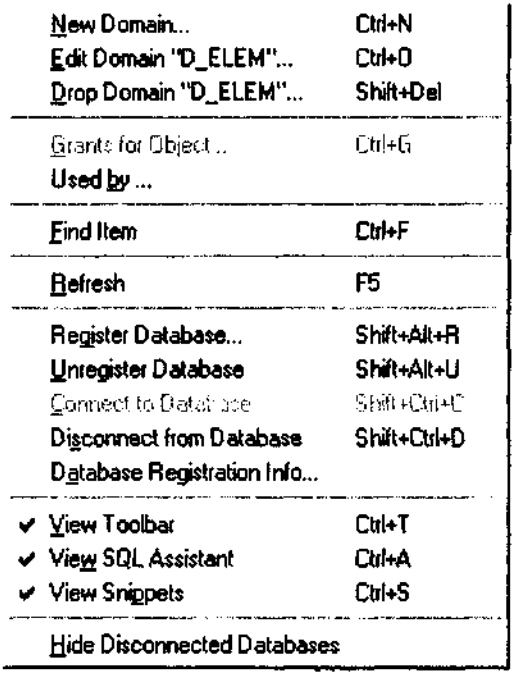


Рис. 11.4. Меню редактирования объектов (доменов).

РЕДАКТОРЫ В QUICKDESK

В QuickDesk предусмотрены редакторы для всех основных типов объектов.

Редактор доменов

Этот редактор активирует окно описания домена. В главной части задается имя домена, тип данных, длина, если это необходимо, признак допустимости Null значений, используемый набор символов и способ сравнения.

В нижней части окна предусмотрена панель для ввода описания (комментария) домена. Затраты на ввод комментариев не так велики, а потери от их отсутствия в дальнейшем могут быть весьма значительными.

При вводе домена можно в диалоге указать значения по умолчанию, контрольные соотношения, а для массивов - их размерность, выбрав требуемую страницу нажатием соответствующей кнопки.

Кроме того, все редакторы объектов предусматривают выдачу итоговой команды создания объекта. Доступ к соответствующей команде DDL осуществляется кнопкой DDL. В окне DDL можно просматривать сформированную в диалоге команду. При необходимости текст команды можно скопировать в SQL Editor и использовать для последующей коррекции и исполнения.

При нажатии кнопки OK сформированный (отредактированный) объект записывается в базу.

Редактор таблиц

Активирует окно описания таблиц. Окно предусматривает выбор страниц для редактирования следующих подобъектов:

- **Fields** - список полей таблицы.
- **Constraints** - список ограничений (первичные и внешние ключи, CHECK-ограничения).
- **Indices** - список индексов таблицы.
- **Dependenencies** - дерево объектов, связанных с текущей таблицей.
- **Triggers** - дерево триггеров для таблицы.
- **Data** - редактор данных (содержимого) таблицы.
- **Descriptions** - редактор текстового описания таблицы (комментарии).
- **DDL** - просмотр исходного текста команды DDL создания таблицы (CREATE TABLE, ALTER TABLE).

Редактор полей содержит список всех полей таблицы с их полным описанием. Двойным щелчком левой клавишей мыши можно перейти в режим редактирования поля. Редактор поля по своему виду аналогичен редактору домена. При щелчке правой клавишей мыши выдается меню:

- New Field - создать новое поле в таблице;
- Edit Field <имя поля> - редактировать выбранное поле;
- Drop Field <имя поля> - удалить выбранное поле;
- Reorder Fields - изменить порядок следования полей в таблице.

При выборе пунктов создания и редактирования полей запускается Редактор полей, о котором уже говорилось. Изменение порядка полей осуществляется стрелками в высвечиваемом окне.

Страница Constraints содержит список всех ограничений таблицы. При щелчке правой клавишей мыши выдается меню:

- New Constraint переводит редактор в режим создания ограничений;
- Drop Constraint удаляет выделенное ограничение;
- Edit Constraints - редактировать ограничение (только в версии 2.00).

В режиме создания ограничений выдается трехстраничное окно для создания ограничений типа Primary Key, Foreign Key, CHECK. Для каждого из этих типов ограничений выдается свое окно, в котором предлагается диалоговый ввод ограничений выбранного типа.

Страница Indices содержит список всех индексов данной таблицы. При щелчке правой клавишей мыши выдается меню:

- New Index переводит редактор в режим создания нового индекса;
- Edit Index <Имя индекса> переводит редактор в режим редактирования указанного индекса;
- Drop Index <Имя индекса> удаляет индекс.

Для перехода в режим редактирования индекса можно также использовать двойной щелчок левой клавишей мыши.

Для редактирования (создания) индексов выдается окно, в котором предлагается диалог для ввода или коррекции индексов.

Просмотр связанных объектов активирует окно со списком объектов, связанных с данной таблицей. Выбрав интересующий объект двойным щелчком левой клавишей мыши можно перейти в редактор выбранного объекта для его просмотра и, при необходимости, изменения.

Редактор триггеров выдает список триггеров, связанных с таблицей. Находясь в этом списке, по щелчку правой клавишей мыши можно выйти в стандартное меню: New Trigger, Edit Trigger <Имя триггера>, Drop Trigger <Имя триггера>. В режим редактирования можно перейти также двойным щелчком левой клавишей мыши. Работа в самом редакторе триггера описана ниже.

Редактор данных высвечивает содержание таблицы либо в табличной форме, когда одной записи таблицы соответствует одна строка, либо в виде формы, когда каждому полю соответствует строка формы. Переход из одного режима в другой осуществляется нажатием кнопки внизу окна (Grid View, Form View). В редакторе данных можно просматривать содержимое таблицы и корректировать сами данные. Кроме того, можно экспортировать данные в файлы. Поддерживаются форматы Excel, Word, RTF, HTML, TXT, CSV, DIF, SYLK, LaTeX. Можно также экспортировать содержимое файла в виде скрипта, содержащего команды *insert*.

Редактор описания таблицы (комментарии) позволяет ввести в базу текстовое описание таблицы для документирования структуры базы.

Кроме того, как и во всех основных редакторах объектов, предусматривается выдача итоговых команд создания объекта на странице DDL-редактора, в данном случае - команд CREATE TABLE, ALTER TABLE. В окне DDL можно просматривать сформированную в диалоге команду. При необходимости текст команды можно скопировать в SQL Editor и использовать для последующей коррекции и исполнения.

При нажатии кнопки OK сформированный (отредактированный) объект записывается в базу.

Из редактора таблиц можно также перейти по кнопкам на инструментальной панели:

- в редактор прав доступа (по кнопке Grants for Table);
- в экспорт данных (Export Data);
- в экспорт данных для загрузки в другую базу в виде скрипта (Export as INSERT statements); скрипт-файл представляет собой множество команд SQL insert, обеспечивающих загрузку в базу содержимого соответствующей таблицы;
- просмотр-печать отчета о структуре таблицы.

Редактор обзоров

Этот редактор активирует окно описания обзора. Окно предусматривает выбор страниц для редактирования следующих подобъектов:

- **SQL** - редактор исходного текста команды SQL создания обзора (CREATE VIEW).
- **Fields** - список полей обзора.
- **Dependencies** - просмотр связанных объектов.
- Triggers - дерево триггеров обзора.
- **Data** - редактор данных (содержимого) обзора.
- **Description** - редактор описания обзора (комментарии).

Редактор исходного текста обзора предназначен для ввода команды CREATE VIEW. Редактор предлагает "заготовку" текста команды, включающую все ее синтаксические элементы.

Список полей обзора аналогичен редактору списка для таблиц, с той лишь разницей, что состав полей полностью определяется конструкцией SELECT команды создания обзора CREATE VIEW.

Просмотр связанных объектов для обзора полностью аналогичен просмотру связанных объектов для таблиц.

Редактор триггеров для обзора полностью аналогичен редактору триггеров для таблиц.

Редактор данных (содержимого) обзора полностью аналогичен редактору данных для таблиц.

Редактор описания обзора (комментарии) полностью аналогичен редактору описания для таблиц.

Редактор хранимых процедур

Активирует окно описания хранимых процедур, которое предусматривает выбор страниц для редактирования следующих подобъектов:

- **Edit** - редактор исходного текста команды SQL создания хранимой процедуры.
- **Description** - редактор описания хранимой процедуры (комментарии).
- **Dependencies** - просмотр связанных объектов.
- **Operations** - просмотр отдельных команд хранимой процедуры.
- **Parameters** - редактор параметров хранимой процедуры.
- **Performance Analysis** - вывод результатов работы процедуры.

Редактор исходного текста команды SQL создания хранимой процедуры содержит текст программы хранимой процедуры, а при создании новой процедуры - синтаксическую заготовку, общую для всех процедур. С точностью до требований синтаксиса хранимых процедур, работа с ним аналогична работе с редакторами исходного текста для объектов других типов. Отличие состоит лишь в том, что таблицы, домены, обзоры можно создать без использования SQL-редакторов, а для хранимых процедур и триггеров их использование обязательно. Созданная в редакторе процедура должна быть откомпилирована. Компиляция осуществляется нажатием кнопки Compile Procedure на инструментальной панели. Выполнить процедуру можно нажатием кнопки Execute Procedure на инструментальной панели. Для анализа результатов выполнения можно использовать *Performance Analysis.*

Редактор описания хранимой процедуры (комментарии) полностью аналогичен редактору описания для таблиц.

Просмотр связанных объектов для хранимых процедур полностью аналогичен просмотру связанных объектов для таблиц.

Редактор отдельных команд хранимой процедуры позволяет просмотреть отдельные операторы SQL, составляющие процедуру.

Редактор параметров хранимой процедуры позволяет просмотреть список всех параметров процедуры и ввести к ним, при необходимости, требуемые комментарии.

Окно Performance Analysis показывает использование процедурой системных ресурсов, интенсивность работы с базой и ряд других данных, полезных для оценки эффективности работы процедуры.

Помимо этого можно использовать и специальные кнопки на панели инструментов, позволяющие:

- задать права на используемые процедурой таблицы (кнопка Grants for Table);
- экспортировать результаты работы в форматах Excel, Word, RTF, HTML, TXT, CSV, DIF, SYLK, LaTeX (кнопка Export Data);
- искать текст (кнопка Find, продолжение поиска - клавиша F3);
- включить отладчик хранимых процедур (кнопка Debug Procedure).

Редактор триггеров

Активирует окно описания триггеров, которое предусматривает выбор страниц для редактирования следующих подобъектов:

- **Trigger** - редактор тела триггера.
- **Description** - редактор описания триггера (комментарии).
- **Dependencies** - просмотр связанных объектов.
- **DDL** - просмотр исходного текста команды SQL создания триггера.

Редактор тела триггера предназначен для ввода команд SQL, реализующих триггер. В окне редактора выводятся команды, обрамляющие тело триггера. В заголовочной части предусмотрены поля ввода для указания:

- имени триггера (Name);
- его типа (Type), выбираемого из выпадающего списка *{before insert, before update, before delete, after insert, after update, after delete);*
- имени таблицы, для которой создается триггер (For Table), выбираемого из выпадающего списка таблиц и обзоров;
- позиции триггера в списке (Position), определяющей порядок включения триггеров в случае, когда для одного объекта предусмотрено несколько триггеров;
- признака активности триггера (Is Active).

Редактор описания триггера (комментарии) полностью аналогичен редактору описания для таблиц.

Просмотр связанных объектов для триггера полностью аналогичен просмотру связанных объектов для таблиц.

Кроме того, как и во всех основных редакторах объектов предусматривается выдача итоговых команд создания объекта, в данном случае - команд CREATE TRIGGER, ALTER TRIGGER. Доступ к соответствующей команде DDL осуществляется кнопкой DDL. В окне DDL можно просматривать сформированную в диалоге команду. При необходимости

текст команды можно скопировать в SQL Editor и использовать для последующей коррекции и исполнения.

На инструментальную панель вынесены кнопки:

- **Compile Trigger** - компиляция и запись триггера в базу;
- **Find** - поиск текста, продолжение поиска - клавиша F3;
- **Save Description** - сохранение в базе описания (комментарий) триггера;
- **Печать метаданных** - просмотр и распечатка документации на триггер.

Редакторгенераторов

Этот редактор активирует окно описания генератора. Окно включает поля имя генератора (Name) и текущее значение генератора (Value). Запись генератора в базу осуществляется кнопкой **Compile.**

Редакторисключений

Активирует окно описания исключений, которое предусматривает выбор страниц для редактирования следующих подобъектов:

- **Exception** - редактор текста исключения.
- **Dependencies** - просмотр связанных объектов.
- **Description** - редактор описания исключения (комментарии).
- **DDL** - просмотр исходного текста команды SQL создания исключения.

Страница редактора текста исключения (Exception) содержит поля для ввода имени исключения (Name) и его текста (Text).

Редактор описания исключения (комментарии) полностью аналогичен редактору описания для таблиц.

Просмотр связанных объектов для исключения полностью аналогичен просмотру связанных объектов для таблиц.

Кроме того, как и во всех основных редакторах объектов предусматривается выдача итоговых команд создания объекта, в данном случае - команды CREATE EXCEPTION. Доступ к соответствующей команде DDL осуществляется кнопкой DDL. В окне DDL можно просматривать сформированную в диалоге команду. При необходимости текст команды можно скопировать в SQL Editor и использовать для последующей коррекции и исполнения.

На инструментальную панель вынесены кнопки:

- **Compile Exception** - компиляция и запись исключения в базу;
- **Save Description** - сохраняет в базе описание (комментарий) исключения.

Редактор функций пользователя

Активирует окно описания функций пользователя, которое предусматривает выбор страниц для редактирования следующих подобъектов:

- **UDF** - редактор описания функции пользователя.
- **Dependencies** - просмотр связанных объектов.
- **Description** - редактор текстового описания функции пользователя (комментарии).
- **DDL** - просмотр исходного текста команды SQL создания описания функции пользователя.

Страница редактора описания функции пользователя содержит поля, задающие имя (Name) функции пользователя, ее точку входа в библиотеке (Entry Point), имя библиотеки (Library Name), тип возвращаемого значения (Returns), способ (By) возврата (по значению или по ссылке) и список типов входных параметров (Input Parameters). Добавление и удаление параметров в списке осуществляется по кнопкам **Add** и **Remove.** Поскольку синтаксис UDF не требует введения названия параметров, то при добавлении параметра необходимо указать только его тип: integer, double precision и т.д.

Просмотр связанных объектов для исключения полностью аналогичен просмотру связанных объектов для таблиц.

Редактор текстового описания функции пользователя (комментарии) полностью аналогичен редактору описания для таблиц.

Кроме того, как и во всех основных редакторах объектов предусматривается выдача итоговых команд создания объекта, в данном случае - команды DECLARE EXTERNAL FUNCTION. Доступ к соответствующей команде DDL осуществляется кнопкой DDL. В окне DDL можно просматривать сформированную в диалоге команду. При необходимости текст команды можно скопировать в SQL Editor и использовать для последующей коррекции и исполнения.

На инструментальную панель вынесены кнопки:

- **Compile Exception** - компиляция и запись описания функции пользователя в базу;
- **Save Description** - сохранение в базе текстового описание (комментария) функции пользователя.

Редактор ролей

Активирует окно управления пользователями и ролями *(User Manager).* Работа с редактором ролей может быть инициирована также из главного меню. Описание работы приведено ниже.

МЕНЮ OPTIONS

Меню Options содержит следующие пункты:

Environment Options Активирует окно настройки опций окружения, позволяющее выбрать режимы работы QuickDesk. К этим режимам относятся:

- порядок сохранение изменений, вносимых в базу;
- перечень оперативно выдаваемой на экран информации;
- внешний вид выдаваемых данных;
- используемые при работе шрифты;
- уровень изоляции для используемых транзакций;
- оформление выдаваемых в табличном виде результатов, включая просмотр таблиц базы данных и результатов выборок из базы.

Editor Options Активирует окно настройки опций редактирования, позволяющее выбрать схемы подготовки данных при работе с QuickDesk. К этим режимам относятся: управление контролем синтаксиса вводимых данных, управление их размещением в окне редактирования, настройка цветовой гаммы, включая возможности выделения синтаксических единиц, управление шрифтами, схемами контроля и использованием клавиатурных шаблонов.

Keyboard **Templates** Активирует окно настройки опций управления клавиатурными шаблонами. В рамках QuickDesk предусмотрено несколько десятков шаблонов, позволяющих вместо набора всего текста на SQL использовать сокращения. Такие сокращения могут быть полезны не только для уменьшения ввода. Их использование гарантирует синтаксическую правильность вводимых конструкций SQL. При желании вдобавок к имеющимся шаблонам можно добавить собственные. Шаблоны можно удалять и редактировать.

Plugins Options Активирует окно настройки опций работы с плагинами. Данная возможность ориентирована на опытных пользователей. Предварительно плагины должны быть подготовлены. Для подключения необходимо выбрать подготовленный плагин, после чего он появляется в меню **Plugins,** из которого уже может быть запущен на выполнение.

МЕНЮ TOOLS

Меню Tools содержит следующие пункты:

SQL Editor Активирует окно редактора текстов SQL. SQL Editor представляет собой полностью настраиваемый, гипертекстовый редактор, позволяющий работать с неограниченным числом запросов. Переключение между запросами осуществляется по пронумерованным закладкам в нижней части окна. Отредактированные запросы сохраняются между сеансами.

В редакторе предусмотрен механизм синтаксических подсказок с возможностью множественного выбора и фильтрации содержания. Можно также использовать настраиваемые клавиатурные сокращения для ввода синтаксических конструкций SQL (см. *Keyboard Templates).*

Предусматриваются также визуальные средства форматирования запросов и информативные сообщения об ошибках.

Результаты выполнения работ можно просмотреть в графическом виде. Предусмотрен механизм поиска в тексте запроса (кнопка Find на инструментальной панели).

В верхней половине окна SQL-редактора осуществляется набор текста команд SQL. Нижняя часть окна предназначена для вывода дополнительной информации, например сообщений об ошибках, плане выполнения запросов. Запуск команд на выполнение удобнее всего осуществлять с использованием инструментальной панели. Результаты выполнения запроса выдаются в табличном виде или в виде формы по желанию пользователя. Переключение осуществляется соответствующей кнопкой. В редакторе SQL можно хранить несколько запросов. Перечень хранимых запросов можно увидеть на панели внизу окна ввода текста. Для упрощения процедуры формирования запросов можно воспользоваться визуальным построителем запросов (Query builder), который можно вызвать как из меню, так и кнопкой на инструментальной панели. Следует заметить, что не всякий запрос может быть построен в рамках Query builder, но его текст можно использовать как заготовку для дальнейшего ввода. В рамках редактора SQL можно при желании использовать сформированные клавиатурные шаблоны.

По выполнении запроса можно просмотреть "историю" выполнения запроса, включая сведения об использованных запросом ресурсах и времени его выполнения. Кроме того, полученные результаты можно сохранить в виде файла. Поддерживаются следующие форматы: Excel, RTF (Word), Текст, HTML, SYLK, DIF, CSV, LaTeX.

New SQL Editor Активирует дополнительное окно редактора текстов SQL. Это позволяет иметь одновременно несколько открытых редакторов SQL.

Query builder Активирует окно интерактивного построителя запросов на выборку данных.

Окно Query builder состоит из четырех основных панелей. Левая верхняя панель предназначена для представления используемых для выборки таблиц. Левая нижняя панель позволяет задать условия выборки, перечень выбираемых полей, порядок группировки данных и их сортировку. Правая верхняя панель содержит перечень доступных таблиц. Правая нижняя панель содержит перечень доступных хранимых процедур.

Построение запроса включает следующие этапы:

- Выбор перечня используемых таблиц и процедур. Выбор состоит в "перетаскивании мышью" необходимых объектов из правых панелей в верхнюю левую. На панели появляются окошки с таблицами с перечислением их полей.
- Установка связей между выбранными объектами. Для установки связи необходимо выделить поле в одной из таблиц и, не отпуская левую клавишу мыши, выбрать нужное поле другой таблицы. После отпускания клавиши на экране появляется линия связи. Если запрос осуществляется из одной таблицы, то этот этап пропускается.
- Выборка перечня выбираемых полей. Выборка полей осуществляется их пометкой в списках полей таблиц.
- При необходимости с помощью нижней панели задаются дополнительные условия выборки, вычисляемые поля, группировки и сортировки результатов.

Если никаких дополнительных требований к запросу нет, то нажав кнопку **Result,** можно увидеть результаты выборки.

Для просмотра и редактирования полученной команды **Select** можно воспользоваться клавишей **Edit.** Для просмотра того, какие ресурсы использовались запросом, можно воспользоваться клавишей **Performance Analysis.**

Полученные результаты можно экспортировать в файл в желаемом формате.

SQL Monitor Активирует окно мониторинга работы с базой данных. В окне можно отследить всю последовательность действий, реально выполняемых в базе, включая подготовительные действия, запуск и завершение транзакций.

SQL Script Активирует окно редактора SQL-скриптов. В рамках редактора SQL Script можно подготовить и записать SQL-скрипт, выполнить ранее подготовленный скрипт, в том числе и подготовленный вне QuickDesk. Кроме того, из окна SQL Script можно запустить поиск текстов в метаданных.

Search in Metadata Активирует окно поиска текстов в метаданных текущей базы. При поиске указывается разыскиваемый текст и типы объектов, в которых он встречается. В результате поиска выдается перечень объектов, удовлетворяющих условиям поиска. Выбрав интересующий объект, можно сразу же перейти к его детальному просмотру и редактированию.

Extract Metadata Активирует окно выборки метаданных из базы. В окне предлагается указать, какой тип данных подлежит выборке: сами метаданные или данные конкретных таблиц. После задания типов выбираемых данных задается имя выходного файла. Далее в диалоге формируется перечень экспортируемых объектов.

Метаданные экспортируются в виде SQL-скриптов, содержащих команды создания соответствующих объектов, собственно данные - в виде последовательности команд **Insert.**

В процессе диалога задаются также режимы формирования скриптов.

Print Metadata Активирует окно печати метаданных. В верхней панели предлагается выбрать тип печатаемых объектов: домены *(Domains),* таблицы *(Tables),* обзоры *(Views),* хранимые процедуры *(Procedures),* триггеры *(Triggers),* исключения *(Exeptions),* пользовательские функции *(UDF).* После выбора типа объектов в левой части окна выводится список объектов данного типа, в правой - список объектов, выбранных для печати. С помощью клавиш **Add, Add all,** Remove, Remove **all** выделенные (все) объекты можно переместить в правое окно или удалить из него. Таким образом, последовательно формируется список объектов, сведения о которых (метаданные) должны быть распечатаны.

С помощью клавиш **Preview** и **Print** можно просмотреть внешний вид документов и распечатать их. Распечатка содержит либо полное описание данных, либо затребованную часть описания. Состав выводимых данных можно задать на основе выборки из перечня описателей, выдаваемых при щелчке правой кнопкой мыши на объекте в правой части окна печати метаданных. В перечень распечатываемых характеристик входят поля *(Fields),* ограничения *(Constraints),* индексы *(Indices),* перечень объектов, зависящих от данного *(Dependent Objects),* перечень объектов, от которых зависит данный *(Dependent On Objects),* параметры (для процедур) объекта *(Parameters),* DDL SQL описание объекта *(DDL),* комментирующее описание *(Description).* Внешний вид выводимых данных позволяет непосредственно включать их в документацию по базе данных.

Распечатка отдельных объектов базы возможна также в режиме их просмотра и редактирования. В соответствующих объектах есть кнопка печати. Там же можно и провести, при желании, модификацию внешнего вида печатаемых документов.

User Manager Активирует окно управления пользователями и ролями. Используя User Manager, можно создавать новые роли или удалять существующие, создавать новых пользователей базы, редактировать данные о существующих пользователях и при необходимости удалять их. Выполнение этих функций осуществляется через кнопки **Add** и **Delete** для работы с ролями и кнопки **Add, Edit, Delete** для работы со списком пользователей. После нажатия кнопки высвечивается соответствующее диалоговое окно, подлежащее заполнению. Для каждого нового пользователя необходимо задать, как минимум, его имя и пароль.

Grant Manager Активирует окно управления правами пользователей и объектов базы данных. Прямое использование команд GRANT и REVOKE в большинстве случаев, хотя и несложно, но крайне неудобно из-за необходимости выдачи их для большой группы объектов. Кроме

того, трудно отслеживать, каким пользователям предоставлены права на объекты, а каким нет. Grant Manager позволяет быстро выполнить эту работу и обеспечивает возможность легко отслеживать права всех пользователей.

Окно Grant Manager содержит две основных панели:

- Слева - панель пользователей, содержащая их полный перечень.
- Справа - панель объектов базы с указанием прав доступа выделенного пользователя на объекты базы. Для каждого "права" выделен столбец **(SEL** - select, **UPD** - update, **DEL** - delete, **EXEC** - execute, **REF** - references). Если право предусмотрено, то в таблице соответствующая ячейка выделена. Предоставление и удаление прав в Grant Manager сводится к пометке ячеек таблицы. Процедура несложная, а главное, наглядная, что позволяет оперативно и качественно следить за состоянием прав пользователей базы. При желании выдаваемый список объектов может фильтроваться по отдельным видам объектов и по наличию или отсутствию прав пользователя на объекты.

Localize IB Message Активирует окно локализации сообщений, выдаваемых InterBase. Локализованные сообщения сохраняются в файле и могут использоваться в сеансах работы QuickDesk.

Localize QuickDesk Активирует окно локализации сообщений, выдаваемых самим QuickDesk. Локализованные сообщения сохраняются в файле и могут использоваться в сеансах работы QuickDesk.

Report Manager Активирует окно создания собственных отчетов для работы с QuickDesk. Использование Report Manager предполагает опыт работы с пакетом FastReport.

МЕНЮ SERVICES

Это меню содержит следующие пункты:

- **Backup Database** Активирует окно для создания архивной копии базы данных. При архивации и восстановлении *(Restore Database)* происходит не только копирование данных, но и удаление устаревших версий записей базы. Последнее позволяет несколько уменьшить размер базы и ускорить работу с ней. Но даже если бы это было не так, регулярное копирование базы необходимо для обеспечения сохранности данных при разного рода аварийных ситуациях. Процесс архивирования осуществляется в диалоговом режиме, позволяющем в зависимости от конкретных потребностей минимизировать время работы. Необходимо помнить, что эта функция работают только с Interbase 6.0/FireBird.

- **Restore Database** Активирует окно для восстановления базы данных (или создания новой копии базы данных) по ее архиву, полученному выполнением команды *Backup Database*. Только для Interbase 6.0/FireBird.
- **Database Statistic** Активирует окно для получения статистики по работе с базой данных. Только для Interbase 6.0/FireBird.
- **Database Properties** Активирует окно для получения сведений о характеристиках базы данных и их модификации.

МЕНЮ PLUGINS

Содержит перечень подключенных программ, если таковые имеются, в противном случае пункт недоступен. Содержимое пунктов определяется реально подключенными плагинами. Формирование списка доступных плагинов осуществляется через пункт *Plugins Options* меню *Options*.

МЕНЮ WINDOWS

Меню содержит следующие пункты:

Windows List Выводит список ранее открытых окон в панели DB Inspector. Для перехода к нужному окну следует выбрать его из полученного списка.

Close All Закрывает все ранее открытые окна.

Close Закрывает все ранее открытые окна для объектов выбранного в подменю типа, а именно

Tables

Views
Procedures
Triggers
Generators
Exeptions
UDF

МЕНЮ HELP

Содержит следующие пункты:

What's new? Содержит дополнительные сведения о текущей версии и ее отличиях от предшествующих.

Contents Содержит упорядоченные по разделам сведения о работе с QuickDesk.

Interbase Help Содержит краткие сведения по работе с InterBase.

Additional help Если доступен, содержит сведения по дополнительным объектам, подключенным к QuickDesk.

Пункты меню **QuickDesk Home Page** и **Send Bug-reports to** предназначены для связи с разработчиками QuickDesk через Internet.

About QuickDesk Содержит краткие сведения о текущей версии QuickDesk.

Необходимо отметить, что файл справки, поставляемый с QuickDesk версий младше 2, оставляет желать лучшего, хотя интуитивно понятный интерфейс позволяет обходиться в большинстве случаев без файла помощи.

В последних версиях Help (английский) значительно расширен, кроме того, в состав меню Help версий 2 включен также текст Help для InterBase 6 (SQL Reference), что придает системе подсказок вполне достойный вид.

ПРИЛОЖЕНИЕ А

Справочник по командам и функциям SQL

А.1. Команды

ALTER DATABASE
ALTER DOMAIN
ALTER EXCEPTION
ALTER INDEX
ALTER PROCEDURE
ALTER TABLE
ALTER TRIGGER
BASED ON
BEGIN DECLARE SECTION
CLOSE
CLOSE (BLOB)
COMMIT
CONNECT
CREATE DATABASE
CREATE DOMAIN
CREATE EXCEPTION
CREATE GENERATOR
CREATE INDEX
CREATE PROCEDURE
CREATE SHADOW
CREATE TABLE
CREATE TRIGGER
CREATE VIEW
DECLARE CURSOR
DECLARE CURSOR (BLOB)
DROP EXCEPTION
DROP EXTERNAL FUNCTION
DROP FILTER
DROP INDEX
DROP PROCEDURE
DROP SHADOW
DROP TABLE
DROPTRIGGER
DROP VIEW
END DECLARE SECTION
EVENT INIT
EVENT WAIT
EXECUTE
EXECUTE IMMEDIATE
EXECUTE PROCEDURE
FETCH
FETCH (BLOB)
GRANT
INSERT
INSERT CURSOR (BLOB)
OPEN
OPEN (BLOB)
PREPARE
REVOKE
ROLLBACK

DECLARE EXTERNAL FUNCTION	SELECT
DECLARE FILTER	SET DATABASE
DECLARE STATEMENT	SET GENERATOR
DECLARE TABLE	SET NAMES
DELETE	SET STATISTICS
DESCRIBE	SET TERM
DISCONNECT	SET TRANSACTION
DROP DATABASE	UPDATE
DROP DOMAIN	WHENEVER

ALTER DATABASE

ОПИСАНИЕ

ALTER DATABASE добавляет новые (вторичные) файлы к существующей базе. Вторичные файлы полезны для контроля роста и расположения базы. Они позволяют распределить файлы базы по устройствам хранения (но должны находиться на том же узле сети, что и первичный файл).

Команда доступна только для создателя базы и SYSDBA и выполняется только в монопольном режиме.

СИНТАКСИС

```
ALTER {DATABASE / SCHEMA}
ADD <add_clause>;

<add_clause>;:=
FILE 'filespec' [<fileinfo>] [<add_clause>]

<fileinfo> = LENGTH [=] int [PAGE [S]] / STARTING [AT [PAGE]] int
```

Таблица A.1. *Синтаксические конструкции команды ALTER DATABASE*

Аргумент	*Описание*
SCHEMA	Альтернативное имя для DATABASE
ADD FILE 'filespec'	Добавляет один или несколько вторичных файлов для получения страниц базы данных после того, как первичный файл заполнен. Для удаленной базы данных вторичные файлы должны быть расположены на том же узле, что и первичный
LENGTH [=] int [PAGE[S]]	Определяет диапазон страниц для вторичного файла, задавая число страниц в каждом файле

Аргумент	Описание
STARTING [AT [PAGE]] int	Определяет диапазон страниц для вторичного файла, задавая стартовый номер страницы

ПРИМЕР

```
ALTER DATABASE
ADD FILE 'testbase.gd1'
       STARTING AT PAGE 10001
              LENGTH 10000
       ADD FILE 'testbase.gd2'
              LENGTH 10000;
```

ALTER DOMAIN

ОПИСАНИЕ

ALTER DOMAIN позволяет изменить описание домена. Изменены могут быть любые описания, кроме типа данных и ограничения NOT NULL. Изменения в домене автоматически сказываются на всех данных, которые на него ссылаются, если соответствующие конструкции домена не были явно переопределены на уровне описания таблиц.

Для изменения типа данных необходимо удалить его (DROP DOMAIN) и вновь создать с новыми характеристиками (CREATE DOMAIN).

Команда доступна только для создателя домена и SYSDBA.

СИНТАКСИС

```
ALTER DOMAIN name {
[SET DEFAULT {literal / NOLL / USER}]
| [DROP DEFAULT]
       | [ADD [CONSTRAINT] CHECK
(<dom_search_condition>)]
       | [DROP CONSTRAINT]
       };

<dom_search_condition> = {
VALUE <operator> <val>
/ VALUE [NOT] BETWEEN <val> AND <val>
       / VALUE [NOT] LIKE <val> [ESCAPE <val>7
       / VALUE [NOT] IN (LIST_<val>)
```

```
        / VALUE IS [NOT] NOLL
        / VALUE [NOT] CONTAINING <val>
        / VALUE [NOT ] STARTING [WITH] <val>
        / (<dom_search_condition>)
        / NOT <dom_search_condition>
        / <dom_search_condition> OR <dom_search_condition>
        j <dom_search_condition> AND <dom_search_condition>

}

<operator> = {= |< |> / <= |>= / !< / !> /<> / != }
```

Таблица А.2. *Синтаксические конструкции команды ALTER DOMAIN*

Аргумент	*Описание*
Name	Имя существующего домена
SET DEFAULT	Указывает значение, которое будет присвоено столбцу, если не было прямого присвоения. Возможные значения: literal - указанная строка, число или дата; NULL - задание значения NULL; USER - имя пользователя, создающего строку. Задаваемые значения должны быть совместимы по типу с типом данных домена. Значения по умолчанию могут быть переопределены при описании столбцов
DROP DEFAULT	Удаляет значения по умолчанию
ADD [CONSTRAINT] CHECK <dom_search_cond>	Добавляет CHECK-ограничение в домен. Домен может иметь только одно CHECK-ограничение (см. CREATE DOMAIN)
DROP CONSTRAINT	Удаляет CHECK-ограничение из описания домена

ПРИМЕР

Пусть ранее был создан домен

```
CREATE DOMAIN PRMKEY
AS INTEGER NOT NULL CONSTRAINT CHECK(value>=0);
```

Тогда его изменение примет вид

```
ALTER DOMAIN PRMKEY drop CONSTRAINT;
ALTER DOMAIN PRMKEY ADD CONSTRAINT CHECK(value>=0);
```

ALTER EXCEPTION

ОПИСАНИЕ

ALTER EXCEPTION изменяет текст сообщения об ошибке.

Команда доступна только для создателя EXCEPTION и SYSDBA.

СИНТАКСИС

```
ALTER EXCEPTION name 'message';
```

Таблица А.3. *Синтаксические конструкции команды ALTER EXCEPTION*

Аргумент	*Описание*
name	Имя существующего исключения
'message'	Строка в кавычках, задающая текст исключения

ПРИМЕР

```
ALTER EXCEPTION ERR_RUBRIC
  'Неверно указана рубрика для книги';
```

ALTER INDEX

ОПИСАНИЕ

ALTER INDEX активирует или деактивирует существующий индекс.

Деактивация и реактивация индекса полезны при вводе большого числа строк. Первый шаг: деактивация индекса. Второй шаг: ввод данных при отключенном индексе. В этом случае не возникает необходимость перестроения индекса и скорость ввода увеличивается. Третий шаг: активация индекса; индекс однократно перестраивается.

Если индекс используется, команда ALTER INDEX блокируется до его освобождения.

ALTER INDEX неприменима для индексов, определенных для ограничений UNIQUE, PRIMARY KEY или FOREIGN KEY. В этом случае выдается сообщение об ошибке. Для изменения таких индексов их необходимо сначала удалить (DROP INDEX), а затем пересоздать (CREATE INDEX).

Команда доступна только для создателя индекса и SYSDBA.

СИНТАКСИС

ALTER INDEX name {ACTIVE / INACTIVE};

Таблица А.4. *Синтаксические конструкции команды ALTER INDEX*

Аргумент	***Описание***
name	Имя существующего индекса
ACTIVE	Активирует индекс
INACTIVE	Деактивирует индекс

ПРИМЕР

```
ALTER INDEX TREADER_RDNAME INACTIVE;
ALTER INDEX TREADER_RDNAME ACTIVE;
```

ALTER PROCEDURE

ОПИСАНИЕ

ALTER PROCEDURE полностью заменяет существующую хранимую процедуру, включая список параметров и тело.

Синтаксис ALTER PROCEDURE совпадает с синтаксисом CREATE PROCEDURE с точность до замены первого слова в команде - CREATE на ALTER.

При изменении структуры параметров необходимо помнить, что это может привести к нарушению работоспособности программ, использующих данную процедуру.

Если процедура используется, команда ALTER PROCEDURE блокируется до ее освобождения.

Изменения в процедуре становятся доступными сразу же после выполнения COMMIT.

Команда доступна только для создателя процедуры и SYSDBA.

СИНТАКСИС

```
ALTER PROCEDURE name
[(LIST_<parameter>)] [RETURNS ((LIST_<parameter>)
AS <procedure_body> [terminator]

<parameter> ::= param <datatype>
```

Таблица А.5. *Синтаксические конструкции команды ALTER PROCEDURE*

Аргумент	***Описание***
Name	Имя существующей процедуры

Аргумент	*Описание*
param <datatype>	Входные параметры процедуры (подробнее см. CREATE PROCEDURE)
RETURNS param <datatype>	Выходные параметры процедуры (подробнее см. CREATE PROCEDURE)
<procedure_body>	Тело процедуры, включая объявление локальных переменных и блок операторов SQL для процедур и триггеров. Подробнее см. CREATE PROCEDURE
terminator	Ограничитель, определенный в команде ISQL SET TERM для пометки конца текста процедуры

ПРИМЕР

```
SET TERM !! ;
ALTER PROCEDURE PAUTHOR (
    P1 CHAR(1),
    P2 CHAR(1))
RETURNS (
    AUTHOR INTEGER,
    AUNAME VARCHAR(60))
AS
begin
 for select author, auname from tauthor
  into :author, :auname
 do
  if(auname>p1) then
  if(auname<p2) then suspend;
end ! !

SET TERM ; !!
```

ALTER TABLE

ОПИСАНИЕ

ALTER TABLE модифицирует структуру существующей таблицы. Одной командой ALTER TABLE можно выполнить несколько операций удаления (DROP) и дополнения (ADD), а начиная с 6 версии и изменения столбца.

Имена ограничения для столбцов необязательны, однако полезны для упрощения работы по их изменению и обработке сообщений об ошибках. Если они не указаны, InterBase создаст их автоматически.

Выполнение ALTER TABLE завершится аварийно, если текущие данные в таблице нарушают добавляемые к таблице определения ограничений первичного (PRIMARY KEY) или уникального (UNIQUE) ключа.

Также нельзя удалить столбец, если он:

- является частью первичного (PRIMARY KEY), уникального (UNIQUE) или внешнего (FOREIGN KEY) ключа;
- используется в контрольном (CHECK) ограничении;
- используется в выражении для вычисляемого столбца.

Для удаления такого столбца необходимо сначала удалить зависящие от него ограничения и вычисляемые столбцы и только потом удалить сам столбец таблицы. Первичные ключевые (PRIMARY KEY) и уникальные (UNIQUE) ограничения не могут быть удалены, если есть ограничения внешнего ключа (FOREIGN KEY). В этих случаях следует удалить ограничение внешнего ключа перед удалением используемого им первичного (PRIMARY KEY) или уникального (UNIQUE) ключа.

При изменении столбца, основанного на домене, может быть установлено дополнительное ограничение контроля (CHECK) для столбца, не отменяющее доменную конструкцию CHECK.

Изменения в таблицах, которые содержат ограничения контроля (CHECK) с подзапросами, также могут быть отклонены.

При изменении или удалении столбца, содержащиеся в нем данные теряются.

Таблица может быть изменена ее создателем и SYSDBA-пользователем.

СИНТАКСИС

```
ALTER TABLE table LIST_<operation>;

<operation>::= {ADD <col_def> / ADD <table_constraint>
  / DROP col / DROP CONSTRAINT constraint
  / ALTER [COLUMN] column_name <alt_col_clause>
}

<col_def>::=
  col {<datatype> / [COMPUTED [BY] (<expr>) / domain}
  [DEFAULT {literal / NULL / USER}]
  [NOT NULL] [<col_constraint>] [COLLATE collation.]

<col_constraint>::=
   [CONSTRAINT constraint] <constraint_def>

<constraint_def>::=
```

```
   {PRIMARY KEY / UNIQUE / CHECK (<search_condition>)
    / REFERENCES other_table [(LIST_other_col)] }

<datatype>::=
   {{SMALLINT / INTEGER / FLOAT / DOUBLE PRECISION}
     [<array_dim>]
    |{DECIMAL / NUMERIC} [(precision [, scale])]
     [<array_dim>]
    / {DATE / TIME / TIMESTAMP} [<array_dim>]
    | {CHAR / CHARACTER / CHARACTER VARYING / VARCHAR}
     [(1...32767)] [<array_dim>] [CHARACTER SET charname]
    / {NCHAR / NATIONAL CHARACTER / NATIONAL CHAR} [VARYING]
     {(1...32767)}[<array_dim>]
    / BLOB {SUB_TYPE {int / subtype_name} ]
     [SEGMENT SIZE int] {CHARACTER SET charname]
    / BLOB {(seglen {, subtype})}
   }

<array_dim>::= [x:y [, x:y ...]]  скобки здесь - часть синтаксиса, а не метаязыка

<table_constraint>::=
    CONSTRAINT constraint <tconstraint_opt>

<tconstraint_opt>::=
   {{PRIMARY KEY / UNIQUE} (LIST_col)
    | FOREIGN KEY (LIST_col) REFERENCES other_table
    / CHECK (<search_condition>)
   ;

<alt_col_clause>::= {TO new_col_name
       | TYPE new_col_datatype
       | POSITION new_col_position}

<new_col_datatype>::= <datatype>

<new_col_name>::= <col>
```

Замечание. Конструкция **ALTER** [**COLUMN**] column_name <alt_col_clause> допустима только в версиях, начиная с 6.

Типы данных TIME | TIMESTAMP допустимы только в версиях, начиная с 6.

См. также CREATE TABLE.

Таблица А.6. *Синтаксические конструкции команды ALTER TABLE*

Аргумент	*Описание*
table	Имя изменяемой таблицы
<operation>	Операция, выполняемая с таблицей. Допустимы операции: ADD - создать новый столбец или табличное ограничение, DROP - удалить существующий столбец или табличное ограничение
<col_def>	Описание вновь добавляемого столбца. Должно включать имя столбца и тип данных. Может включать значения по умолчанию, ограничения столбца и вид упорядочения (COLLATE)
<table_constraint>	Описание нового табличного ограничения. Добавить можно только одно табличное ограничение
col	Имя добавляемого, изменяемого или удаляемого столбца. Имена должны быть уникальны в пределах таблицы
<constraint>	Имя добавляемого или удаляемого ограничения в пределах таблицы. Имена должны быть уникальны в пределах таблицы
COLLATE collation	Добавляет вид упорядочения для заданной таблицы
<datatype>	Тип данных для добавляемого столбца
domain	Имя домена, на который опирается описание столбца
COMPUTED . [BY] <expr>	Описывает вычисляемый столбец. Конструкция <expr> задает порядок вычисления; <expr> может быть любым допустимым в SQL выражением, возвращающим единственное значение (массивы и BLOB недопустимы). Все столбцы, участвующие в вычислении, должны существовать на момент ввода вычисляемого столбца
NOT NULL	Указывает, что столбец не может содержать значение NULL. Если таблица содержит строки, то кроме NOT NULL следует задавать и значение по умолчанию (см. DEFAULT)

Аргумент	*Описание*
DEFAULT	Устанавливает значение по умолчанию, присваиваемое столбцу, если его значение не установлено явно. Возможные значения: literal - указанная строка, число или дата; NULL - задание значения NULL; USER - имя пользователя, создающего строку. Задаваемые значения должны быть совместимы по типу с типом данных столбца. Значение по умолчанию для столбца переопределяют значения, заданные на доменном уровне
<constraint_def>	Ограничения на столбец
CONSTRAINT	Задает имя ограничения столбца
DROP CONSTRAINT	Удаляет из таблицы указанное ограничение
new_col_position	Число, задающее новый номер столбца в таблице

ПРИМЕР

```
ALTER TABLE TBOOK ADD CHECK(BOOKNM NOT IN ('Слон',
'Моська'));
```

Добавляются ограничение в таблицу.

```
ALTER TABLE TBOOK
 ALTER COLUMN BOOKNM POSITION 2,
 DROP CONSTRAINT INTEG_20;
```

Меняется порядок столбцов в таблице и удаляется CHECK ограничение. Имя ограничению присваивалось системой, поэтому перед удалением его следует найти. В конкретном случае оно будет другим.

Отметим, что не всякая синтаксически правильная команда ALTER TABLE может быть выполнена, например, из-за нарушений логической целостности, недопустимых преобразований типов данных и т.п.

ALTER TRIGGER

ОПИСАНИЕ

ALTER TRIGGER изменяет описание существующего триггера. Если какие-либо аргументы для ALTER TRIGGER пропущены, то они будут

приняты по значениям, указанным в CREATE TRIGGER или в последней из команд ALTER TRIGGER.

С помощью ALTER TRIGGER можно изменить:

- только информацию заголовка триггера, включая состояние активности, условие запуска и номер триггера в последовательности выполнения триггеров для данной таблицы;
- только тело триггера, то есть набор команд триггера, которые следуют за предложением AS;
- заголовок и тело триггера. В этом случае новое определение триггера заменяет старое.

Триггер может быть изменен его создателем и SYSDBA-пользователем.

Чтобы заменить триггер, созданный автоматически конструкцией CHECK, следует использовать команду ALTER TABLE.

СИНТАКСИС

```
ALTER TRIGGER name
[ACTIVE / INACTIVE]
    [{BEFORE / AFTER} {DELETE / INSERT / UPDATE}]
    /"POSITION number]
    [AS <trigger_body>7 [terminator]
```

Таблица А.7. *Синтаксические конструкции команды ALTER TRIGGER*

Аргумент	*Описание*
name	Имя существующего триггера
ACTIVE	Указывает, что триггер активен (по умолчанию)
INACTIVE	Указывает, что триггер не активен
BEFORE	Указывает, что триггер выполняется перед заданной операцией
AFTER	Указывает, что триггер выполняется после заданной операцией
DELETE | INSERT | UPDATE	Указывает операцию, инициирующую триггер
POSITION number	Задает номер триггера. Номер – целое в диапазоне 0–32767 (по умолчанию - 0). Триггеры на одну и ту же операцию выполняются в порядке возрастания номеров. Если имеется несколько триггеров с одним номером, то порядок их выполнения не определен

Аргумент	Описание
<trigger_body>	Тело триггера - блок операторов (инструкций) на языке SQL для триггеров и хранимых процедур (подробнее см. CREATE TRIGGER)
terminator	Ограничитель, определенный в команде ISQL SET TERM для пометки конца текста процедуры. Не нужен, если меняется только заголовок триггера

ПРИМЕР

```
ALTER TRIGGER I_TBOOK_1 INACTIVE;
```

Деактивирует триггер. Описание триггера сохраняется, но включаться он не будет, пока не выполнится команда

```
ALTER TRIGGER I_TBOOK_1 ACTIVE;
```

```
SET TERM !! ;
ALTER TRIGGER I_TBOOK_1
as
begin
  if (new.UNIKEY is NULL) then
   new.UNIKEY=GEN_ID(sysnumber,1);
  if (new.MATHERKEY is NULL or new.MATHERKEY<0) then
   exception NO_RUBRIC;
  if (new.MATHERKEY>0) then
   if (NOT EXISTS (select * from TBOOK
                   where (unikey=new.MATHERKEY)))
   then exception ERR_RUBRIC;
  if (new.BOOKNM is NULL) then exception NO_BOOKNM;
endSET TERM ; !!
```

Заменяется тело триггера.

BASED ON

ОПИСАНИЕ

BASED ON - директива препроцессора и соответственно используется только внутри программы на базовом языке. BASED ON создает переменную базового языка, основанную на определении столбца. Переменная базового языка наследует атрибуты, описанные для столбца, и характеристики, которые делают тип переменной совместимым с исполь-

зуемым языком программирования. Использование конструкции BASED ON позволяет гарантировать соответствие типов данных в программе типам в базе данных, при изменении типа в базе достаточно просто обработать исходный текст препроцессором и перетранслировать программу. Например, в С команда BASED ON прибавляет один байт к переменным CHAR и VARCHAR, чтобы разместить признак конца - символ *ЧУ*.

BASED ON применяется в разделе определения переменных программ и не требует ключевых слов EXEC SQL.

Для объявления переменной базового языка, достаточно большой, чтобы хранить сегмент BLOB при выполнении операций FETCH, следует использовать опцию SEGMENT предложения BASED ON. Размер переменной определяется, исходя из длины сегмента столбца BLOB. Если длина сегмента для столбца BLOB изменена в базе данных, необходимо перетранслировать программу, чтобы скорректировать размер переменных базового языка, созданных с помощью BASED ON.

Может использоваться в SQL.

СИНТАКСИС

BASED *[***ON***]* *[*dbhandle./table.col*[*.**SEGMENT***/* variable;

Таблица А.8. *Синтаксические конструкции директивы BASED ON*

Аргумент	*Описание*
Dbhandle.	Дескриптор базы данных, в которой находится таблица для программ, работающих с несколькими базами данных. dbhandle должен быть предварительно объявлен в инструкции SET DATABASE
table.col	Имена таблицы и столбца, на котором строится переменная
SEGMENT	Задает размер локальной переменной на основе длины сегмента столбца BLOB во время операций BLOB FETCH. Используется только, когда table.col относится к столбцу данных типа BLOB
variable	Имя переменной базового языка, наследующей характеристики столбца таблицы базы данных

ПРИМЕР

```
EXEC SQL
        BEGIN DECLARE SECTION;
BASED ON TBOOK.UNIKEY wUNIKEY;
BASED ON TBOOK.MOTHERKEY wMOTHERKEY;
BASED ON TBOOK.BOOKNM wBook;
```

```
EXEC SQL
      END DECLARE SECTION;
```

Для программ на C или C++ в **результате обработки препроцессором** объявления этих переменных примут **вид (для столбцов тестовой базы):**

```
int wUNIKEY;
int wMOTHERKEY;
char wBook [251] ;
```

BEGIN DECLARE SECTION

ОПИСАНИЕ

BEGIN DECLARE SECTION используется в приложениях с внедренным (embedded) SQL, чтобы идентифицировать начало объявлений переменных базового языка для переменных, которые будут использоваться в последующих инструкциях SQL. BEGIN DECLARE SECTION является также директивой препроцессора, которая дает указание gpre объявить переменную SQLCODE, в которой затем будут размещаться коды возврата исполняемых команд SQL.

BEGIN DECLARE SECTION должна всегда появляться в разделе объявления глобальных переменных модуля.

Может использоваться в SQL.

СИНТАКСИС

BEGIN DECLARE SECTION;

ПРИМЕР

Следующие инструкции внедренного SQL объявляют раздел **и** переменную базового языка:

```
EXEC SQL
BEGIN DECLARE SECTION;
 int wUNIKEY, wMOTHERKEY;
 char wBook [251];
EXEC SQL
END DECLARE SECTION;
```

См. также END DECLARE SECTION и **BASED ON.**

CLOSE cursor

ОПИСАНИЕ

CLOSE закрывает указанный открытый курсор, освобождая строки в его активном наборе и все связанные с ним ресурсы системы. Курсор - односторонний указатель в упорядоченном наборе строк, полученных по выражению select в инструкции DECLARE CURSOR. Курсор обеспечивает последовательный доступ к выбранным строкам и модификации их на месте.

Может использоваться в SQL.

СИНТАКСИС

```
CLOSE cursor;
```

Таблица А.9. *Синтаксические конструкции команды CLOSE cursor*

Аргумент	*Описание*
Cursor	Имя открытого курсора

Имеется четыре связанных с этим выражений для курсора.

Таблица АЛО. *Команды работы с курсором*

Этап	*Оператор*	*Действие*
1	DECLARE CURSOR	Объявляет курсор. Инструкция SELECT определяет строки, выбираемые для курсора
2	OPEN	Выбирает строки, указанные для поиска в DECLARE CURSOR. Полученные строки становятся активным набором курсора
3	FETCH	Получает текущую строку из активного набора, начиная с первой строки. Последующие команды FETCH продвигают курсор по набору
4	CLOSE	Закрывает курсор и освобождает ресурсы системы

Команды FETCH выполняются только при открытом курсоре. Пока курсор не закрыт и не открыт повторно, InterBase не обрабатывает изменения в базе, сделанные другими пользователями. Другими словами курсор видит то состояние базы, которое было на момент его открытия. Что-

бы получить доступ к изменениям, сделанным другими пользователями необходимо закрыть и повторно открыть курсор.

Кроме CLOSE автоматически закрывают все курсоры в транзакции команды COMMIT и ROLLBACK.

ПРИМЕР

```
EXEC SQL
DECLARE Mcursor CURSOR FOR
SELECT ...;
. . .

EXEC SQL
OPEN Mcursor;
. . .
// Открытие цикла чтения
. . .
EXEC SQL
FETCH Mcursor INTO . . . ;
// Проверка на конец выборки и выход из цикла
. . .
// Окончание цикла чтения
EXEC SQL
      CLOSE Mcursor;
```

CLOSE blob_cursor

ОПИСАНИЕ

CLOSE закрывает открытый для чтения или вставки курсор BLOB. Вообще курсор BLOB должен быть закрыт только после:

- Выборки всех сегментов BLOB для операций чтения BLOB (BLOB READ).
- Вставка всех сегментов для операций BLOB INSERT.

Может использоваться в SQL.

СИНТАКСИС

```
CLOSE blob_cursor;
```

Таблица **A.11.** *Синтаксические конструкции команды CLOSE blob_cursor*

Аргумент	*Описание*
blob_cursor	Имя открытого курсора BLOB

ПРИМЕР fefc

Следующая команда внедренного SQL закрывает BLOB курсор.

```
EXEC SQL
     CLOSE BC;
```

COMMIT

ОПИСАНИЕ

COMMIT используется для завершения транзакции и записи всех изменений в базу. Делает изменения видимыми для команд в последовательностях транзакций SNAPSHOT и READ-COMMITTED.

Закрывает открытые курсоры, если параметр RETAIN не используется.

Транзакция, завершенная COMMIT, считается завершенной успешно.

Для завершения транзакций, открытых по умолчанию, всегда следует задавать COMMIT или ROLLBACK.

После транзакций read-only, то есть не порождающих изменений, лучше использовать COMMIT, а не ROLLBACK. Результат один и тот же, но затраты времени и места следующими транзакциями уменьшаются.

Аргумент RELEASE сохранен только для обеспечения совместимости с предыдущими версиями InterBase.

СИНТАКСИС

COMMIT *[***WORK***]* *[***TRANSACTION** name*]* *[***RELEASE***]* *[***RETAIN** *[***SNAPSHOT***]*/ ;

Таблица А.12. *Синтаксические конструкции команды COMMIT*

Аргумент	*Описание*
WORK	Используется только для совместимости с другими реляционными базами, требующими его
TRANSACTION name	Сохраняет в базе имя транзакции. Без этого режима COMMIT воздействует на транзакцию по умолчанию
RELEASE	Используется только для совместимости с ранними версиями InterBase
RETAIN *[SNAPSHOT]*	Сохраняет изменения и текущий операционный контекст

ПРИМЕР

Следующая команда фиксирует изменения в базе:

```
COMMIT;
```

Следующая команда фиксирует изменения в базе поименованной транзакции:

```
COMMIT TR1;
```

Следующая команда использует COMMIT RETAIN, чтобы сохранить изменения при поддержании текущего операционного контекста. В этом случае с точки зрения пользователя можно считать, что все изменения записаны, но транзакция как была активной, так и осталась. Все команды, как данной транзакции, так и других будут видеть данные такими же, как, если бы COMMIT RETAIN не выдавалась. В то же время при откатах после выполнения COMMIT RETAIN будет восстанавливаться состояние на момент выдачи последнего COMMIT RETAIN. Использование COMMIT RETAIN вместо пары COMMIT - SET TRANSACTION позволяет несколько ускорить обработку данных и влечет задержку разрешения доступа к измененным транзакцией данным.

```
COMMIT RETAIN;
```

CONNECT

ОПИСАНИЕ

КомандаCONNECT:

- Инициализирует структуры данных базы данных.
- Определяет, находится ли база данных на том же (локальная база данных) или на другом узле (удаленная база). Базы данных, используемые клиентами Windows 3.1, - всегда на удаленных серверах. Сообщение об ошибках возникает, если InterBase не может найти базу данных.
- Самостоятельно определяет имя пользователя и пароль для использования при подключении приложений к базе данных. Клиенты Windows 3.1 должны всегда посылать правильное имя пользователя и пароль.
- Соединяется с базой данных и проверяет страницу заголовка. Файл базы данных должен содержать корректную базу данных, и номер версии дисковой структуры (on-disk structure - ODS) базы данных должен быть распознан установленной версией InterBase на сервере, иначе InterBase возвращает ошибку.

При соединении с базой данных CONNECT использует заданный по умолчанию набор символов (NONE) или указанный в предыдущей команде SET NAMES. Каждый раз, когда используется CONNECT, чтобы присоединиться к базе данных, предыдущее соединение разрывается.

СИНТАКСИС

```
CONNECT ['] <filespec> ['] {USER 'username' [PASSWORD
'password']};
```

В версиях, начиная с 6, синтаксис несколько расширяется

```
CONNECT 'filespec' {USER 'username'} {PASSWORD 'password'}
        [CACHE int] {ROLE 'rolename'];
```

Синтаксис во внедренном SQL

```
CONNECT [TO]
  {ALL / DEFAULT} <config_opts>
  / LIST_<specif>;

<specif>::= <db_specs> <config_opts>

<db_specs>::= dbhandle
        / {'filespec' / :variable} AS dbhandle
<config_opts>::= {USER {'username' / :variable}]
        {PASSWORD {'password' / :variable}]
        {ROLE {'rolename' / :variable}]
        [CACHE int [BUFFERS 7 7
```

Таблица А.13. *Синтаксические конструкции команды CONNECT*

Аргумент	*Описание*
filespec	Имя файла базы данных. Может включать спецификацию пути и узла
USER 'username'	Строка, которая определяет имя пользователя, используемое при соединении с базой данных. Сервер проверяет имя пользователя по isc.gdb-базе данных защиты. Имя пользователя на сервере нечувствительно к регистру
PASSWORD 'password'	Строка, которая определяет пароль, используемый при соединении с базой данных. Сервер проверяет имя пользователя и пароль по isc.gdb-базе данных защиты. Пароль на сервере чувствителен к регистру

Аргумент	*Описание*
CACHE int	Устанавливает количество буферов кэша для базы данных, определяющее количество страниц базы, одновременно доступных программе. Если не указано, применяется значение по умолчанию (256).
ROLE 'rolename'	Задает роль, с которой пользователь соединяется с базой (см. команды CREATE ROLE, GRANT).
dbhandle	Дескриптор базы данных, с которой выполняется соединение (см. команду SET DATABASE).
variable	Имя базовой переменной, содержащей соответствующую символьную строку.

ПРИМЕР

Следующая команда открывает базу данных.

```
CONNECT 'testbase.gdb' USER 'SYSDBA' PASSWORD 'masterkey';
```

Открытие с указанием роли выглядит так

```
CONNECT 'testbase.gdb' USER 'SYSDBA' PASSWORD 'masterkey'
 ROLE 'BIBL';
```

Открытие в программе на базовом языке с предварительным объявлением базы выглядит так

```
EXEC SQL
SET DATABASE MYBASE = "testbase.gdb";
EXEC SQL
CONNECT MYBASE;
```

CREATE DATABASE

ОПИСАНИЕ

CREATE DATABASE создает базу данных и устанавливает для нее следующие характеристики:

- Имя первичного файла, который идентифицирует базу данных для пользователей. По умолчанию, базы данных содержатся в одном файле.
- Имена любых вторичных файлов, в которых размещается база данных. База данных может постоянно находиться в более чем одном файле на диске, если дополнительные имена файла оп-

ределены как вторичные файлы. Если база данных создана на удаленном сервере, вторичные спецификации файла не могут включать имя узла.

- Размер страниц базы данных. Увеличение размера страницы может улучшать работу по следующим причинам:
  а) индексы работают быстрее, потому что глубина индексов уменьшается,
  б) хранение длинных строк на одной странице более эффективно,
  в) запись и получение BLOB-данных более эффективно, когда они помещаются на одной странице.

Если большинство транзакций обрабатывает одну или небольшое число строк данных, меньший размер страницы может быть предпочтительнее, так как меньшее количество данных должно быть обработано и меньшее количество памяти будет использовано буфером системы ввода-вывода.

- Число страниц в каждом файле базы данных.
- Набор символов, используемый базой данных.
- Системные таблицы, которые описывают структуру базы данных.

После создания базы данных пользователь может определять ее таблицы, обзоры, индексы и системные обзоры.

При использовании сети NetWare CREATE DATABASE позволяет также определить протокол Write-ahead Log (WAL). Следует отметить, что использование сети NetWare при работе с InterBase нежелательно, поскольку накладывает существенные ограничения на использования многих команд.

СИНТАКСИС

```
CREATE {DATABASE / SCHEMA} 'filespec'

[USER 'username' [PASSWORD 'password']]
      [PAGE_SIZE [=] int]
      [LENGTH [=] int [PAGE [s]]]
      [DEFAULT CHARACTER SET charset]
      [<secondary_file>];

<secondary_file>::= FILE 'filespec' [<fileinfo>] f<secondary_file>]

<fileinfo>: := LENGTH [=] int [PAGE [S]] / STARTING [AT [PAGE]] int [<fileinfo>]
```

Таблица А.14. *Синтаксические конструкции команды CREATE DATABASE*

Аргумент	*Описание*
filespec	Спецификация файла новой базы данных. Соглашения о наименовании файла зависят от платформы (Windows> Linux ...)
USER 'username'	Имя пользователя проверяется на корректность комбинации имени пользователя и пароля в базе данных защиты по серверу, где база данных будет размещена. Клиентские приложения Windows должны указывать имя пользователя при соединении с сервером
PASSWORD 'password'	Пароль проверяется на корректность комбинации имени пользователя и пароля в базе данных защиты по серверу, где база данных будет размещена. Клиентские приложения Windows должны указывать имя пользователя и пароль при соединении с сервером
PAGE_SIZE /=/ int	Размер в байтах страниц базы данных. Допустимо 1024 (по умолчанию), 2048, 4096 или 8192
DEFAULT CHARACTER SET charset	Устанавливает набор символов для базы данных, используемый по умолчанию; charset - имя набора символов в кавычках. Если опущено, то в качестве набора символов по умолчанию принимается NONE
FILE 'filespec'	Имена одного или нескольких вторичных файлов для хранения страниц базы по заполнении первичного. Для баз данных на удаленных серверах спецификация вторичных файлов не должна содержать имя узла
STARTING [AT /PAGE.// int	Определяет стартовый номер страницы для вторичного файла
LENGTH /=/ int [PAGE[S]]	Определяет длину первичного или вторичного файла базы данных. Для первичного файла используется только при одновременном определении вторичного файла в той же самой команде

ПРИМЕР

Создает базу данных в текущей директории.

```
CREATE DATABASE 'testbase.gdb';
```

Создает базу данных со страницей 2048 байт (по умолчанию, 1024).

```
CREATE DATABASE 'testbase.gdb' PAGE_SIZE 2048;
```

Создает базу данных, размещенную в двух файлах, и устанавливает кодовую таблицу, используемую по умолчанию.

```
CREATE DATABASE 'testbase.gdb'
       DEFAULT CHARACTER SET 'WIN1251'
       FILE " testbase.gd1" STARTING AT PAGE 10001 LENGTH
10000 PAGES;
```

CREATE DOMAIN

ОПИСАНИЕ

CREATE DOMAIN создает в базе описание столбца, которое используется как шаблон при описании столбцов таблиц в командах CREATE TABLE или ALTER TABLE. Описание домена содержит следующий набор характеристик:

- Тип данных.
- Необязательное значение по умолчанию.
- Необязательный запрет значений NULL.
- Необязательное контрольное ограничение (CHECK).
- Необязательное предложение порядка сравнения.

Ограничение контроля (CHECK) домена описывает условия <dom_search_condition>, которые должны быть истинными при вводе данных в столбец, базирующийся на описании домена. Ограничение CHECK не может ссылаться на другое описание домена или столбца.

При создании домена необходимо следить за непротиворечивостью ограничений. Например, нельзя задавать конструкции типа DEFAULT NULL NOT NULL.

Спецификация данных типов CHAR, VARCHAR или текстового BLOB может содержать указание набора символов CHARACTER SET, используемого для данных домена. Если CHARACTER SET не указан, то будет использован набор символов, используемый базой по умолчанию.

Конструкция COLLATE специфицирует порядок сравнения данных типов CHAR, VARCHAR или текстового BLOB. Выбор порядка сравнения ограничен допустимыми для набора символами, определенным в базе по умолчанию или явно заданным для домена.

Столбцы, базирующиеся на описании домена, наследуют все его характеристики. Значения по умолчанию, порядок сравнения и установка NOT NULL могут быть переопределены при описании столбцов, бази-

рующихся на описании домена. Ограничение домена CHECK не может быть переопределено. При указании в описании столбца конструкции CHECK она добавляется к конструкции CHECK, указанной для домена.

Замечание. Если при описании столбцов таблицы домен не указывается явно, то автоматически строится новый домен и его описание заносится в таблицу описаний доменов (RDB$FIELDS).

СИНТАКСИС

```
CREATE DOMAIN domain [AS] <datatype>
[DEFAULT {literal / NOLL / USER} ]
       /NOT NULL] /CHECK (<dom_search_condition>)]
       /COLLATE collation7;
```

Конструкция COLLATE неприменима для BLOB столбцов.

```
<datatype>::= {
{SMALLINT / INTEGER / FLOAT / DOUBLE PRECISION} /<ar-
ray_dim> 7
       / {DECIMAL / NUMERIC} /(precision /, scale7)7
[<array_dim>7
       / {DATE / TIME / TIMESTAMP} [<array_dim>]
       / {CHAR / CHARACTER / CHARACTER VARYING / VARCHAR}
            [(1...32767)] [<array_dim>] /CHARACTER SET
charname7
       / {NCHAR / NATIONAL CHARACTER / NATIONAL CHAR}
            [VARYING] [(1...32767)] /<array_dim> 7
       / BLOB [SUB_TYPE {int / subtype_name}] [SEGMENT
SIZE n]

/CHARACTER SET charname]
       / BLOB [(seglen [, subtype7)7
       }

<array_dim>::= [LIST_<dim>]
```

Здесь квадратные скобки являются ***частью синтаксиса описания массивов, а не элементом метаязыка.***

```
<dim>::= x[:y]
<dom_search_condition>::= {
```

```
VALUE <operator> <val>
       / VALUE [NOT] BETWEEN <val> AND <val>
       / VALUE [NOT] LIKE <val> [ESCAPE <val>J
       / VALUE [NOT] IN (<val> [, <val> ...])
       / VALUE IS [NOT J NULL
       / VALUE [NOT] CONTAINING <val>
       / VALUE [NOT 7 STARTING [WITH] <val>
       / (<dom_search_condition>)
       / NOT <dom_search_condition>
       / <dom_search_condition> OR <dom_search_condition>

/ <dom_search_condition> AND <dom_search_condition>
       ;
<operator> = {= / < / > /<= / >= / !< / !> / <> /-'•-}
```

Замечание. Типы данных TIME | TIMESTAMP допустимы только в версиях, начиная с 6.

Таблица А.15. *Синтаксические конструкции команды CREATE DOMAIN*

Аргумент	*Описание*
domain	Уникальное имя создаваемого домена
<datatype>	Тип данных, допустимый в SQL
DEFAULT	Указывает значение, которое будет присвоено столбцу, если не было прямого присвоения. Возможные значения: literal - указанная строка, число или дата; NULL - задание значения NULL; USER - имя пользователя, создающего строку. Задаваемые значения должны быть совместимы по типу с типом данных домена. Значения по умолчанию могут быть переопределены при описании столбцов
NOT NULL	Указывает, что вводимые в столбец данные не могут принимать значение NULL
CHECK (<dom_search_cond>)	Добавляет CHECK-ограничение в домен. Домен может иметь только одно CHECK-ограничение

Аргумент	*Описание*
VALUE	Заменитель имени столбца, базирующегося на домене
COLLATE collation	Определяет последовательность сравнения для домена

ПРИМЕР

Следующая инструкция создает домен, который предназначен для хранения номера месяца. Ключевое слово VALUE заменяет имя столбца, базирующегося на этом домене.

```
CREATE DOMAIN DMONTH AS
SMALLINT
CHECK (VALUE BETWEEN 1 AND 12);
```

Следующая команда ограничивает значения столбцов, основанных на домене, перечнем обозначений химических элементов:

```
CREATE DOMAIN D_ELEM AS
CHAR(2) CHARACTER SET WIN1251
CHECK (value in ('H', 'Li', 'Na', 'K'));
```

Следующая команда создает домен, который определяет массив символьного (CHAR) типа данных:

```
CREATE DOMAIN CHAR5_4_5 AS CHAR(10) [4:5];
```

Следующая команда создает домен с заданием имени пользователя, как значения по умолчанию.

```
CREATE DOMAIN USERNAME AS
VARCHAR(20) CHARACTER SET WIN1251
DEFAULT '***'
COLLATE WIN1251
```

Если в таблице присутствует столбец, основанный на таком домене, то при добавлении в таблицу новой строки в это поле будет автоматически записываться имя пользователя, выполнившего добавление, если поле явно не указано в списке команды INSERT, в противном случае будет записано явно указанное значение. Для того чтобы исключить подобную «фальсификацию», следует присвоение или изменение полей типа USERNAME контролировать в триггерах.

```
CREATE TABLE . . . ( . . . , USERNAME USERNAME, . . . ) ;
```

Следующая команда создает домен для BLOB с текстовым подтипом, который имеет назначенный набор символов:

```
CREATE DOMAIN TEXT_BLOB AS
 BLOB sub_type 0 segment size 80;
```

В этом случае команда создания таблицы без использования доменов

```
CREATE TABLE TBOOK (
 UNIKEY PRMKEY,
 MATHERKEY INTEGER,
 BOOKNM VARCHAR(250) character set WIN1251 collate WIN1251,
 REFERAT BLOB sub_type 0 segment size 80,
 NUM_ALL SMALLINT DEFAULT 0 NOT NULL,
 NUM_PRESENCE SMALLINT DEFAULT 0 NOT NULL);
```

будет идентична команде с использованием домена TEXT_BLOB

```
CREATE TABLE TBOOK (
 UNIKEY PRMKEY,
 MATHERKEY INTEGER,
 BOOKNM VARCHAR(250) character set WIN1251 collate WIN1251,
 REFERAT TEXT_BLOB,
 NUM_ALL SMALLINT DEFAULT 0 NOT NULL,
 NUM_PRESENCE SMALLINT DEFAULT 0 NOT NULL);
```

CREATE EXCEPTION

ОПИСАНИЕ

CREATE EXCEPTION создает исключение - определяемую пользователем ошибку со связанным с ней сообщением. Исключения могут быть инициированы в триггерах и хранимых процедурах.

Исключения глобальны в базе данных. Любое сообщение или набор сообщений исключений доступны всем хранимым процедурам и триггерам в приложении. Например, база данных может иметь английские и русские версии тех же самых сообщений исключения и использовать соответствующий набор, когда необходимо.

Когда исключение инициировано триггером или хранимой процедурой, то прекращается работа триггера или хранимой процедуры, отображается сообщение об ошибках и выполняется откат транзакции.

Исключения могут быть перехвачены и обработаны с помощью конструкции WHEN в хранимой процедуре или триггере (см. CREATE TRIGGER, CREATE PROCEDURE).

СИНТАКСИС

CREATE EXCEPTION name **'message';**

Таблица **A.** 16. *Синтаксические конструкции команды CREATE EXCEPTION*

Аргумент	*Описание*
name	Имя исключения. Должно быть уникальным в перечне исключений базы данных
message	Строка в кавычках, задающая текст исключения. Максимальная длина - 78 символов

ПРИМЕР

```
CREATE EXCEPTION ERR_RUBRIC 'Неверно указана рубрика для
книги';
```

Следующая команда в триггере (хранимой процедуре) инициирует ранее описанное исключение

```
exception NO_RUBRIC;
```

Следующая команда в триггере «бандитничает» с исключением, меняя его текст

```
. . .
if (new.MATHERKEY is NULL or new.MATHERKEY<0) then
 BEGIN
  update RDB$EXCEPTIONS SET
     Rdb$message='Не указана рубрика для <'
       || new.BOOKNM || '>'
   where Rdb$exception_name='NO_RUBRIC';
  exception NO_RUBRIC;
 END
. . .
```

Подобная конструкция основывается на том, что:

- тексты исключений хранятся в системной таблице RDB$EXCEPTIONS;
- выдача команды exception порождает откат транзакции;
- после отката транзакции все внесенные ей изменения, в том числе и в таблице RDB$EXCEPTIONS, отменяются.

CREATE GENERATOR

ОПИСАНИЕ

CREATE GENERATOR объявляет генератор в базе данных и устанавливает в 0 его начальное значение. Генератор выдает последовательные числа, которые можно автоматически вставить в столбец, используя функцию GEN_ID (). Генератор обычно используется, чтобы гарантировать уникальность значений в первичных ключах (PRIMARY KEY), которые должны однозначно идентифицировать связанную строку, если само значение при этом не существенно.

База данных может содержать любое число генераторов. Генераторы глобальны в базе данных и могут использоваться и модифицироваться в любой транзакции. InterBase не повторяет свои значения во время выполнениятранзакции.

После того как генератор создан, можно изменить его текущее значение, используя команду SET GENERATOR.

Генератор можно использовать в триггерах, хранимых процедурах и SQL операторах, вызывающих функцию GEN_ID().

Замечание. В ряде случаев можно рекомендовать использование в базе одного генератора, тогда все сгенерированные с его помощью ключи будут уникальными. Поскольку значение, выдаваемое генератором, лежит в диапазоне от -2^{31} до $2^{31}-1$, то даже, если отказаться от отрицательных величин, можно получить более 2000000000 значений. Поэтому опасения, что в вашей базе за все время ее существования окажется большее количество строк, явно преувеличены. В любом случае базы объемом более нескольких сотен гигабайт потребуют какой-то иной платформы.

СИНТАКСИС

```
CREATE GENERATOR name;
```

Таблица А.17. *Синтаксические конструкции команды CREATE GENERATOR*

Аргумент	***Описание***
name	Имя генератора

ПРИМЕР

Создаются генератор SYSNUMBER.

Значение, формируемое генератором, используется в триггере вставки (BEFORE INSERT) для таблицы TBOOK, обеспечивая задание уникального значения первичного ключа UNIKEY, если он не задан явно.

```
CREATE GENERATOR SYSNUMBER;

. . .

CREATE TRIGGER I_TBOOK_1 FOR TBOOK
ACTIVE BEFORE INSERT POSITION 0
as
begin
 if (new.UNIKEY is NULL) then
  new.UNIKEY=GEN_ID(sysnumber,1);
. . .
```

CREATE INDEX

ОПИСАНИЕ

Команда CREATE INDEX создает индекс для таблиц базы данных. Индексы используются для ускорения доступа к данным. Использование индексов в конструкции WHERE может значительно увеличить скорость поиска.

Столбцы BLOB и массивы не могут использоваться в индексе.

Уникальный ключ (UNIQUE) не может быть создан по столбцу, который уже содержит повторяющиеся или NULL значения.

ASC или DESC указывает на порядок сортировки в индексе. Для более быстрого получения ответов на запросы, требующие отсортированных результатов, следует использовать индекс в соответствии с порядком сортировки, указанным в конструкции ORDER BY. ASC и DESC индексы могут быть построены одновременно по одному и тому же столбцу или набору столбцов для доступа к данным в разном порядке.

Замечание. Наличие индекса ускоряет поиск данных, но замедляет их изменение, поскольку при изменении данных надо модифицировать не только строки таблиц, но и всю совокупность индексов, связанных с таблицей. Для оптимизации использования индексов можно применить SET STATISTICS, позволяющую определить частоту их использования, перестроить индекс, делая его активным или неактивным с последовательными вызовами ALTER INDEX. При массовом внесении изменений можно рекомендовать следующую последовательность действий: деактивация индекса, ввод данных, активация индекса, при которой выполняется перестройка индексов (см. ALTER INDEX).

СИНТАКСИС

```
CREATE [UNIQUE] [ASC[ENDING] / DESC[ENDING]]
INDEX index ON table (LIST_col);
```

Таблица А.18. *Синтаксические конструкции команды CREATE INDEX*

Аргумент	*Описание*
UNIQUE	Обеспечивает контроль уникальность значений ключа
ASC[ENDING]	Задает сортировку по возрастанию ключа (по умолчанию)
DESC[ENDING]	Задает сортировку по убыванию ключа
index	Задает уникальное в базе имя индекса
table	Задает имя индексируемой таблицы
col	Задает имя столбца в списке, образующем индекс

ПРИМЕР

Создается индекс по фамилиям читателей в таблице:

```
CREATE INDEX TREADER_RDNAME ON TREADER (RDNAME);
```

Создается такой же индекс, но для выборки по фамилиям в обратном алфавитном порядке (по убыванию значений ключа). Для символьных данных эта конструкция экзотична, но для столбцов с данными типа «календарная дата» очень удобна.

```
CREATE DESCENDING INDEX TREADER_RDNAME ON TREADER (RDNAME);
```

Создается уникальный (UNIQUE) индекс по номерам читательских билетов:

```
CREATE UNIQUE INDEX TREADER_RDNUMB ON TREADER (RDNUMB);
```

Создается индекс по двум столбцам:

```
CREATE UNIQUE INDEX MURA ON TREADER (RDNAME, RDNUMB);
```

Замечание. Второй и последний индексы для данной таблицы бессмысленны и приведены только для демонстрации синтаксиса.

CREATE PROCEDURE

ОПИСАНИЕ

CREATE PROCEDURE определяет новую хранимую процедуру в базе данных. Хранимая процедура - отдельная программа, написанная на

SQL для процедур и триггеров InterBase и хранящаяся как часть метаданных базы данных. Хранимые процедуры могут получать входные параметры и возвращать значения вызвавшим их приложениям.

Язык процедур и триггеров InterBase включают все команды SQL манипуляции данными и некоторые расширения, в том числе IF ... THEN ... ELSE, WHILE ... DO, FOR SELECT ... DO, исключения и обработку ошибок.

Имеются два типа процедур. Во-первых, процедуры выборки, которые приложение может использовать вместо таблицы в команде SELECT. Процедура выборки должна быть определена так, чтобы возвращать одно или несколько значений, либо сообщение об ошибке.

Во-вторых, выполнимые процедуры, которые приложение может вызвать непосредственно командой EXECUTE PROCEDURE. Выполнимая процедура может и не возвращать значения вызывающей программе.

Хранимая процедура вне зависимости от типа состоит из заголовка и тела.

Заголовок процедуры содержит:

- имя хранимой процедуры, которое должно быть уникальным среди имен процедур и таблиц в базе данных;
- необязательный список входных параметров и их типов, которые процедура получает от вызывающей программы;
- RETURNS, сопровождаемая списком выходных параметров и их типов, если процедура возвращает значения вызывающей программе.

Тело процедуры содержит:

- необязательный список локальных переменных и их типов;
- блок команд на языке процедур и триггеров InterBase в скобках BEGIN и END. Блок может включать другие блоки, то есть иметь несколько уровней вложения.

Поскольку каждое предложение в теле хранимой процедуры должно завершаться **";",** необходимо задать какой-либо другой символ для завершения ISQL предложения CREATE PROCEDURE. Для этих целей перед командой CREATE PROCEDURE используется команда SET TERM, задающая ограничитель, отличный от **";".** После завершения текста команды CREATE PROCEDURE вновь используется SET TERM для восстановления стандартного ограничителя ";". Если процедура создается из приложения или какой-либо специальной утилиты, то в команде SET TERM нет необходимости, более того, ее появление будет трактоваться, как синтаксическая ошибка.

InterBase не допускает изменения в хранимых процедурах структур данных, т.е. команд типа DROP TABLE, DROP EXCEPTION.

Чтобы видеть все процедуры, определенные для текущей базы данных, или текст и параметры заданной процедуры, используется внутренняя команда ISQL SHOW PROCEDURES или SHOW PROCEDURE.

Описание хранимых процедур записываются в системные таблицы. Текст тела процедуры, описание ее заголовка, кроме описания параметров, хранятся в таблице RDB$PROCEDURES, описания параметров хранятся в таблице RDB$PROCEDURE_PARAMETERS

Язык хранимых процедур и триггеров InterBase является полным языком программирования. Он включает:

- SQL команды манипулирования данными, а именно INSERT, UPDATE, DELETE и singleton SELECT.
- SQL команды и выражения, включая пользовательские функции (UDF - User Defined Functions), подключенные к базе, и генераторы.
- Расширения SQL, включая команды присвоения, управления последовательностью обработки, контекстные переменные (для триггеров), команды инициации событий, исключений и обработки ошибок.

В следующей таблице приведены команды расширения SQL.

Таблица А.19. *Синтаксические конструкции расширения SQL для процедур*

Команда	***Описание***
BEGIN … END	Определяет блок команд рассматриваемых синтаксически как одна команда. Ключевое слово BEGIN помечает начало блока (открывающаяся скобка), ключевое слово END помечает конец блока (закрывающаяся скобка). Ограничитель ";" после BEGIN и END не ставится
variable = expression	Оператор присвоения. Присваивает выражение столбцу, локальной переменной, входному или выходному параметру
/* comment_text */	Комментарий программиста. Может занимать несколько строк
EXCEPTION exception_name	Инициирует указанное исключение. Исключение - это генерируемая пользователем ошибка, которая может быть обработана с помощью оператора WHEN

Команда	*Описание*
EXECUTE PROCEDURE proc_name /Var /, var .../] /RETURNINGJVALUES var /, var .../]	Вызывает на выполнение указанную в proc_name хранимую процедуру со списком входных аргументов после имени процедуры и указанием списка возвращаемых значений, указываемого после ключевых слов RETURNINGJVALUES. Входные и выходные аргументы должны быть переменными, объявленными в процедуре. Допустимы вложенные процедуры и рекурсия
EXIT	Вызывает переход к завершающему процедуру оператору END
FOR <select_statement> DO <compound_statement>	Обеспечивает повторение составного оператора, записанного после ключевого слова DO для каждой из строк, полученных при выполнении запроса, заданного в конструкции <select_statement>. <select_statement> - обычная команда SELECT, за исключением того, что она требует конструкции INTO, завершающей команду
<compound_statement>	Составной оператор - это либо отдельный оператор на языке хранимых процедур и триггеров, либо блок, заключенный в скобки BEGIN и END
IF(<condition>) THEN <compound_statement> /ELSE <compound_statement>7	Проверяет условие <condition> и, если оно принимает значение TRUE (истина), выполняет составной оператор, следующий за ключевым словом THEN, иначе выполняет составной оператор, следующий за ключевым словом ELSE, если оно указано. Условие <condition> представляет собой булевское выражение, принимающее одно из значений: (TRUE, FALSE или UNKNOWN). Обычно условие - это пара выражений, соединенных операцией сравнения (=, !=, <, <= ...)
POST_EVENT event_name	Инициирует событие event_name
SUSPEND	В SELECT-процедуре SUSPEND возвращает вызывающему приложению выходные параметры для каждой строки. Не рекомендуется для выполняемых процедур (т.е. вызываемых по EXECUTE PROCEDURE)
WHILE (<condition>) DO <compound_statement>	До тех пор пока условие <condition> является истиной (TRUE), выполняется составной оператор <compound_statement>. Сначала проверяется условие <condition> и, если оно истинно, то выполняется оператор <compound_statement>. Цикл повторяется, пока условие остается истинным

Команда	*Описание*
WHEN *{LIST_*<error>| ANY*}* DO <compound_statement>	Команда управления обработкой ошибок. Когда возникает ошибка одного из указанных видов, выполняется составной оператор <compound_statement>. Команда WHEN, если она необходима, должна располагаться в конце блока непосредственно перед END. <error>: EXCEPTION exception_name; SQLCODE errcode или GDSCODE number. ANY указывает на необходимость обработки ошибок любых типов

СИНТАКСИС

```
CREATE PROCEDURE name
[(LIST_<lparam>)]
[RETURNS (LIST_<lparam>)]
      AS <procedure_body> [terminator]

<lparam>::= param <datatype>

<procedure_body>::=
[<variable_declaration_list>]
      <block>

<variable_declaration_list>::=
DECLARE VARIABLE var <datatype>;
[<variable_declaration_list>]

<block>::=
BEGIN
<compound_statement>

[<compound_statement> ...]
END

<compound_statement>::= {<block> / statement;}

<datatype>::= {
{SMALLINT / INTEGER / FLOAT / DOUBLE PRECISION}
/ {DECIMAL / NUMERIC} [(precision [, scale])]
/ {DATE / TIME / TIMESTAMP}
```

```
/ {CHAR / CHARACTER / CHARACTERVARYING / VARCHAR } [(int)  ]
[CHARACTER SET charname]
/ {NCHAR / NATIONAL CHARACTER / NATIONAL CHAR}
      [VARYING]  [(int)] }
```

Замечание. Типы данных TIME | TIMESTAMP допустимы только в версиях, начиная с 6.

Таблица А.20. *Синтаксические конструкции команды CREATE PROCEDURE*

Аргумент	*Описание*
name	Имя хранимой процедуры. Должно быть уникальным среди имен процедур, таблиц и обзоров в базе данных
*LIST*_<param <datatype>>	Входные параметры, которые вызывающая программа использует для передачи значений в процедуру. param - имя входного параметра, должно быть уникальным в перечне переменных процедуры. <datatype> - любой допустимый в InterBase тип
RETURNS (*LIST*_<param «datatype»)	Выходные параметры используются для передачи возвращаемых значений в вызывающую программу. param - имя выходного параметра, должно быть уникальным в перечне переменных процедуры. <datatype> - любой допустимый в InterBase тип. Процедура выполняет возврат значений с помощью выходных параметров при выполнении оператора SUSPEND
AS	Ключевое слово, отделяющее заголовок процедуры от ее тела
DECLARE VARIABLE var <datatype>	Объявляет локальные переменные, используемые только в данной процедуре. Каждое объявление должно начинаться с DECLARE VARIABLE и заканчиваться ";". var - имя локальной переменной, уникальное в перечне переменных процедуры. <datatype> - любой допустимый в InterBase тип
statement	Любой простой оператор, допустимый в SQL для хранимых процедур и триггеров. Каждый оператор (кроме блоков BEGIN–END) должен заканчиваться ";"
terminator	Ограничитель, заданный командой SET TERM и указывающий конец тела процедуры. Используется только в ISQL

ПРИМЕР

```
CREATE PROCEDURE PBUTHOR (CODE INTEGER)
RETURNS (AUTHORS VARCHAR (250))
AS
 declare variable auname varchar(60);
 declare variable UNIKEY integer;
 declare variable ws integer;
begin
 WS=-1;
 AUTHORS=' ';
 for select a.UNIKEY, b.auname
  from tbook a, tauthor b, tbook_author c
  where (a.unikey=:Code and a.unikey=c.bookkey and
c.author=b.author)
  into :UNIKEY, :auname
  do
   begin
    if(ws=-1) then authors=auname;
    else authors=authors||', '||auname;
    ws=UNIKEY;
   end
 if(ws!=-1) then suspend;
end
```

CODE - код книги (первичный ключ) в таблице книг

AUTHORS - возвращаемое значение - список авторов книги в виде списка

auname, UNIKEY - рабочие переменные процедуры для считывания первичного ключа книги и фамилии автора соответственно.

ws - рабочая переменная для контроля формирования списка авторов

Вызов процедуры в SELECT с параметром

```
select * from PbUTHOR(12)
```

Результат выборки

Буассо Марк, Деманж Мишель, Мюнье Жан-Мари

А теперь рассмотрим аналогичную процедуру, только без параметров и возвращающую данные по всем книгам с указанием по ним списка авторов. Данная процедура в процессе работы использует предыдущую.

```
CREATE PROCEDURE PBOOKAUTHOR
RETURNS (
```

```
    UNIKEY INTEGER,
    MATHERKEY INTEGER,
    BOOKNM VARCHAR (250),
    AUTHORS VARCHAR (250),
    REFERAT BLOB sub_type 0 segment size 80)
AS
begin
 for select a.UNIKEY, a.matherkey, a.booknm,
    a.referat
  from tbook a
  order by 1
  into :UNIKEY, :matherkey, :booknm, :referat
  do
   begin
    select AUTHORS from PBUTHOR(:UNIKEY) into :AUTHORS;
    suspend;
  end
end
```

Вызов процедуры в SELECT с параметром

```
select booknm, authors from PBOOKAUTHOR
```

Результат выборки

BOOKNM	AUTHORS
Макрокоманды MS Word	Культин Н.Б.
Word 6 for Windows	Хаселир Райнер Г., Фаненштих Клаус
Язык С++	Подбельский Вадим Валериевич
Введение в С++ Builder	Елманова Н.З., Кошель С П.
Borland - Технологии. SQL-Link InterBase, Paradox for Windows, Delphi	Дунаев Сергей
С и С++ Справочник	Луис Дерк
Введение в технологию АТМ	Буассо Марк, Деманж Мишель, Мюнье Жан-Мари
The history of England. Absolute Monarchy.	Бурова И.И.
Справочник по правописанию и литературной правке	Розенталь Д.Э.
Тесты. Сборник 11 класс. Варианты и ответы государственного тестирования. Пособие для подготовки к тестированию.	без авторов ну совсем

BOOKNM	AUTHORS
Математические вопросы динамики вязкой несжимаемой жидкости.	Ладыжинская Ольга Александровна
Кровь нерожденных	Дашкова Полина
Тайна	Хмелевская Иоанна

Поскольку процедура PBUTHOR возвращает в точности одну строку, то вместо команды

```
select AUTHORS from PBUTHOR(:UNIKEY) into :AUTHORS;
```

в процедуре PBOOKAUTHOR можно использовать просто вызов процедуры PBUTHOR

```
execute procedure PBUTHOR :UNIKEY
 RETURNING_VALUES :AUTHORS;
```

CREATE SHADOW

ОПИСАНИЕ

CREATE SHADOW применяется, чтобы избежать потери доступа к базе за счет создания одной или нескольких копий базы на дополнительных устройствах. Каждая копия содержит один или несколько "теневых" файлов, образующих теневой набор. Каждый теневой набор идентифицируется положительным целым числом.

Теневой диск содержит три компоненты:

- Базу данных, для которой создается тень.
- Системную таблицу RDB$FILES, содержащую список теневых файлов и другие данные о базе.
- Теневой набор, содержащий один или несколько файлов тени.

При выдаче CREATE SHADOW тень устанавливается по последней подключенной к приложению базе. Теневой набор состоит из одного или нескольких файлов. В случае отказа диска администратор базы данных может активировать теневой диск так, что он займет место базы данных.

Если указан режим CONDITIONAL, то при активации администратором базы данных теневого диска для замены основной базы создается новая тень. Если размер базы превышает доступное на диске пространство для тени на одном диске, можно использовать режим вторичных файлов <secondary_file>, чтобы задать несколько файлов тени, которые можно распределить по разным дискам.

Чтобы добавить вторичный файл к существующей тени, нужно удалить тень (DROP SHADOW) и пересоздать с помощью CREATE SHADOW с требуемым числом файлов.

СИНТАКСИС

```
CREATE SHADOW set_num [AUTO / MANUAL] [CONDITIONAL]
'filespec' [LENGTH [=] int [PAGE [S]]]
      [<secondary_file>];

<secondary_file>::= FILE 'filespec' [<fileinfo>] [<secon-
dary_file>]

<fileinfo>::= LENGTH [=] int [PAGE [S]  \ STARTING [AT
[PAGE] ] int [<fileinfo>]
```

Таблица A.21. *Синтаксические конструкции команды CREATE SHADOW*

Аргумент	*Описание*
set_num	Положительное целое, идентифицирующее теневой набор, к которому принадлежат все файлы, перечисленные в команде
AUTO	Определяет заданное по умолчанию поведение доступа для баз данных в случае недоступности тени. Все приложения и соединения сохраняются; ссылки к тени удаляются, а теневой файл отключается
MANUAL	Определяет, что соединения с базой данных будут разрываться до тех пор, пока тень не станет доступной или все ссылки на нее не будут удалены из базы данных
CONDITIONAL	Создает новую тень, позволяя продолжить работу, если первичная тень становится недоступной или если тень заменяет базу данных из-за дискового отказа
filespec	Имя файла с указанием пути доступа для файла тени. Спецификация теневого файла не может содержать имени узла
LENGTH [=] int [PAGE[S]]	Размер теневого файла в страницах базы данных. Размер страницы здесь определяется размером страницы самой базы

Аргумент	*Описание*
<secondary_file>	Задает характеристики вторичного файла, включая длину. Используется для первичного файла только, если вторичный файл задается в той же команде
STARTING /AT [PAGE]] int	Задает стартовый номер страницы, с которой начинается вторичный файл тени

ПРИМЕР

Создается автоматическая тень с режимом AUTO для testbase.gdb (один файл).

```
CREATE SHADOW 1 AUTO 'testbase.shd';
```

Создается автоматическая тень с режимом CONDITIONAL для testbase.gdb (один файл).

```
CREATE SHADOW 2 CONDITIONAL 'testbase.shd' LENGTH 1000;
```

Следующие команды создают многофайловые наборы теней для базы данных testbase.gdb. В первом задаются стартовые страницы для файлов тени, количество страниц в файлах тени.

```
CREATE SHADOW 3 AUTO                                    '
              'testbase.sh1'
       FILE 'testbase.sh2'                              \
              STARTING AT PAGE 1000
       FILE 'testbase.sh3'
              STARTING AT PAGE 2000;
CREATE SHADOW 4 MANUAL 'testbase.sh4'
       LENGTH 1000
       FILE 'testbase.sh1'
              LENGTH 1000
       FILE 'testbase.sh2';
```

CREATE TABLE

ОПИСАНИЕ

CREATE TABLE описывает создаваемую таблицу, ее столбцы, ограничения логической целостности в существующей базе. Пользователь, создавший таблицу, является ее владельцем и имеет на нее все привилегии, включая возможность передавать права (GRANT) другим пользователям, триггерам и хранимым процедурам.

CREATE TABLE поддерживает несколько режимов для описания столбцов:

- Локальные столбцы специфицируются именем и типом данных.
- Вычисляемые столбцы основываются на выражениях. Значения столбцов вычисляются при обращении к таблице. Если тип не специфицирован, то InterBase выбирает подходящий.
- Столбцы, базирующиеся на доменах, наследуют все характеристики домена. Кроме того, описания столбца могут содержать новые значения по умолчанию, атрибуты NOT NULL, дополнительные виды контроля (CHECK) и режимы упорядочения, которые дополняют или замещают указанные в описании домена.

Для данных символьных типов CHAR, VARCHAR, текстовый BLOB описание может включать конструкцию CHARACTER SET, специфицирующую указываемый символьный набор для конкретного столбца. Если конструкция CHARACTER SET не указана явно, то используется символьный набор, принятый по умолчанию для базы данных. Если набор символов изменен, все последовательно определенные столбцы будут иметь новый набор символов, набор символов для существующих столбцов при этом не изменится.

Предложение COLLATE задает порядок сравнения символьных данных типов CHAR, VARCHAR и текстов BLOB. Выбор порядка сравнения ограничен в зависимости от набора символов столбца, который является или заданным по умолчанию набором символов для всей базы данных, или набором, определенным в предложении CHARACTER SET как часть определения типа данных.

В частности, если для базы данных указан DEFAULT CHARACTER SET NONE, то нельзя указывать CHARACTER SET для отдельных столбцов, при попытках сортировки данных будут возникать ошибки. Для CHARACTER SET WIN1251 (русский) допустимы COLLATE WIN1251 (по умолчанию) или COLLATE PDXCYRL. В первом случае упорядочение производится в порядке возрастания кодов (так же, как и при CHARACTER SET NONE) - прописная латынь, строчная латынь, прописная кириллица, строчная кириллица, во втором при сортировке порядок не зависит от того, являются ли буквы прописными или строчными.

NOT NULL - атрибут, который предотвращает ввод NULL (пустых) или неизвестных значений в столбце. NOT NULL воздействует на все операции INSERT и UPDATE в столбце.

Ограничения целостности могут быть определены для таблицы при ее создании. Ограничения целостности - это правила контроля базы данных и ее содержание. Они определяют связи "столбец к таблице" и "таблица к таблице" и контролируют правильность ввода данных. Ограниче-

ния целостности охватывают все транзакции обращения к базе данных и автоматически поддерживаются системой.

CREATE TABLE может создавать следующие типы ограничений целостности:

- Ограничения PRIMARY KEY задают уникальные идентификаторы для каждой строки в таблице. Значения в этом столбце или упорядоченном наборе столбцов не могут повторяться в таблице. Столбец (их может быть несколько), входящий в первичный ключ - PRIMARY KEY должен также иметь атрибут NOT NULL. Таблица может иметь только одно ограничение PRIMARY KEY, которое может быть определено на одном или нескольких столбцах.
- Уникальные (UNIQUE) ключи гарантируют, что никакие две строки не имеют то же самое значение в таблице для указанного столбца или упорядоченного набора столбцов. Столбец (их может быть несколько), входящий в уникальный ключ должен также иметь атрибут NOT NULL. Таблица может иметь несколько уникальных (UNIQUE) ключей. Уникальный (UNIQUE) ключ может быть использован в качестве внешнего ключа (FOREIGN KEY) в другой таблице.

Ссылочные ограничения гарантируют, что значения в наборе столбцов, которые определяют внешний ключ (FOREIGN KEY), - те же самые, что и значения в уникальном (UNIQUE) или первичном ключе (PRIMARY KEY) в указанной таблице. Прежде чем будет создано ссылочное ограничение, в таблице, на которую строится ссылка, должен уже быть определен уникальный или первичный ключ.

Ограничения контроля (CHECK) задают условие <search_condition>, которое должно быть истинно для вставок или модификаций к указанной таблице. <Search_condition> может проверять вводимые данные на принадлежность к допустимому диапазону значений или сравнивать их с данными других столбцов, а также с результатами поиска по существующей базе данных.

Для непоименованного ограничения система назначает уникальное имя ограничения, сохраненное в таблице системы RDB$RELATION_CONSTRAINTS.

Конструкция EXTERNAL FILE объявляет, что данные создаваемой таблицы размещаются во внешнем по отношению к базе данных файле (не InterBase). Конструкция EXTERNAL FILE используется для создания таблиц InterBase, основанных на данных внешних источников, в том числе управляемых другими операционными системами или приложениями, а также для передачи данных в существующие таблицы InterBase из внешних файлов.

СИНТАКСИС

```
CREATE TABLE table {EXTERNAL [FILE] 'filespec']
(<col_def> {, <col_def> / <tconstraint> ...]);

<col_def>::= col {datatype / COMPUTED [BY] (<expr>) / do-
main}
[DEFAULT {literal / NULL / USER}]
       [NOT NULL7 {<col_constraint>7
       {COLLATE collation]
```

Конструкция COLLATE неприменима к BLOB столбцам.

```
<datatype>::= {
{SMALLINT / INTEGER / FLOAT / DOUBLE PRECISION} [<ar-
ray_dim>]
       / {DECIMAL / NUMERIC} {(precision {, scale])]
{<array_dim>J
       / {DATE / TIME / TIMESTAMP} [<array_dim>]
       / {CHAR / CHARACTER / CHARACTER VARYING / VARCHAR}
     . {(int)] {<array_dim>] {CHARACTER SET charname J
       / {NCHAR / NATIONAL CHARACTER / NATIONAL CHAR}
             {VARYING] [(int)] [<array_dim>]
       / BLOB {SUB_TYPE {int / subtype_name} ]
 {SEGMENT SIZE int] [CHARACTER SET charname]
       / BLOB {(seglen {, subtype7)7
       ;

<array_dim>::= [LIST_<dim>]
```

Квадратные скобки здесь - это ***часть синтаксиса, а не метаязыка*** (в квадратных скобках задается размерность массива).

```
<dim>::=x[:y]
```

<expr> - любое корректное SQL выражение, возвращающее отдельную величину (BLOB и массив не допускаются).

```
<col_constraint>: := [CONSTRAINT constraint] <con-
straint_def>
```

```
<constraint_def>::= {UNIQUE / PRIMARY KEY
/ REFERENCES other_table [(LIST_other_col)]
{ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
| CHECK (<search_condition>)}

<tconstraint>::= CONSTRAINT constraint <tconstraint_def>
[<tconstraint>]

<tconstraint_def>::= {{PRIMARY KEY / UNIQUE} (LIST_col)
/ FOREIGN KEY (LIST_col) REFERENCES other_table
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
{ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}
/ CHECK (<search_condition>)}

<search_condition>::=
{<val> <operator> {<val> / (<select_one>)}
      | <val> [NOT] BETWEEN <val> AND <val>
      / <val> [NOT] LIKE <val> {ESCAPE <val>
      / <val> [NOT] IN (LIST_<val> / <select_list>)

/ <val> IS [NOT] NULL
      / <val> {{NOT} {= | < | >} | >= |<=}
            {ALL / SOME / ANY} (<select_list>)
      / EXISTS (<select_expr>)
      / SINGULAR (<select_expr>)
      / <val> [NOT] CONTAINING <val>
      / <val> {NOT} STARTING {WITH} <val>
      / (<search_condition>)
      / NOT <search_condition>
      / <search_condition> OR <search_condition>
      / <search_condition> AND <search_condition>}

<val>::= {
col [<array_dim>] j <constant> / <expr> | <function>
       / NULL / USER / RDB$DB_KEY
```

```
        } [COLLATE collation]

<constant>::= num / "string" / charsetname "string"

<function>::= {
COUNT (* / [ALL] <val> / DISTINCT <val>)
        / SUM ( [ALL] <val> / DISTINCT <val>)
        / AVG ( [ALL] <val> / DISTINCT <val>)
        / MAX ( [ALL] <val> / DISTINCT <val>)
 / MIN ( [ALL] <val> / DISTINCT <val>)
        / CAST (<val> AS <datatype>)
        / UPPER (<val>)
        / GEN_ID (generator, <val>)
        }
<operator>::= {= /< /> /<= />= / !< / !> / <> / ! =}
```

<select_one> - SELECT с одним столбцом, возвращающий в точности одно значение.

<select_list> - SELECT с одним столбцом, возвращающий несколько значений.

<select_expr> - SELECT, возвращающий список значений и несколько строк.

Замечание. Типы данных TIME | TIMESTAMP допустимы только в версиях, начиная с 6.

Таблица А.22. *Синтаксические конструкции команды CREATE TABLE*

Аргумент	*Описание*
Table	Имя таблицы. Имя должно быть уникальным внутри базы в списке имен таблиц и процедур
EXTERNAL /FILE/ 'filespec'	Объявляет, что данные создаваемой таблицы размещаются во внешнем по отношению к базе данных файле. **Имя** и полный путь к файлу задаются в filespec
col	**Имя** столбца. Имя должно быть уникальным внутри таблицы в списке имен столбцов

Аргумент	*Описание*
<datatype>	Тип данных для столбца
COMPUTED [BY] (<expr>)	Описывает вычисляемый столбец. Конструкция <expr> задает порядок вычисления. <expr> может быть любым допустимым в SQL выражением, возвращающим единственное значение (массивы и BLOB недопустимы). Все столбцы, участвующие в вычислении, должны существовать на момент ввода вычисляемого столбца
domain	Имя домена, на который опирается описание столбца
COLLATE collation	Задает вид упорядочения для таблицы. Если COLLATE задано также и для базового домена, то данная конструкция переопределяет доменную
DEFAULT	Устанавливает значение по умолчанию, присваиваемое столбцу, если его значение не установлено явно. Возможные значения: literal - указанная строка, число или дата; NULL - задание значения NULL; USER - имя пользователя создающего строку. Задаваемые значения должны быть совместимы по типу с типом данных столбца. Значение по умолчанию для столбца переопределяет значение, заданное на доменном уровне
CONSTRAINT constraint	Задает имя ограничения на таблицу или столбец. Ограничение задает правило контроля или организации данных. Если конструкция не указана, InterBase генерирует системное имя
NOT NULL	Указывает, что столбец не может содержать значение NULL. Если таблица содержит строки, то кроме NOT NULL следует задавать и значение по умолчанию (см. DEFAULT)
references . . .on delete {no action | cascade | set default | set null}	Обеспечивает синхронное изменение внешнего ключа и первичного. No action - не меняет внешний ключ, может повлечь возникновение ошибки, тогда удаление отменяется Cascade - удаляет строки соответствующие внешнему ключу. Set default устанавливает значения во внешнем в соответствии со значениями по умолчанию. Если значение по умолчанию не находится в первичном ключе, то удаление отменяется. Значения по умолчанию устанавливаются при определении ограничений. Дальнейшее их изменение не влияет на значе-

Аргумент	*Описание*
	ния, используемые ограничениями. Set null - значения внешнего ключа устанавливаются в NULL
references . . . on update {no action | cascade | set default | set null}	Обеспечивает синхронное изменение внешнего ключа и первичного. No action - не меняет внешний ключ, может повлечь возникновение ошибки, тогда изменение отменяется Cascade - заменяет значения во внешнем ключе по первичному. Set default устанавливает значения во внешнем в соответствии со значениями по умолчанию. Если значение по умолчанию не находится в первичном ключе, то удаление отменяется. Значения по умолчанию устанавливаются при определении ограничений. Дальнейшее их изменение не влияет на значения, используемые ограничениями. Set null - значения внешнего ключа устанавливаются в NULL

ПРИМЕР

Создается таблица с первичным ключом на уровне столбца

```
CREATE TABLE TBOOK (
  UNIKEY PRMKEY PRIMARY KEY,
  MATHERKEY INTEGER,
  BOOKNM VARCHAR (250),
  REFERAT BLOB sub_type 0 segment size 80,
  NUM_ALL SMALLINT DEFAULT 0 NOT NULL,
  NUM_PRESENCE SMALLINT DEFAULT 0 NOT NULL);
```

Создается таблица с вычисляемыми столбцами и первичным ключом на уровне таблицы.

```
CREATE TABLE TBOOK_AUTHOR (
 UNIKEY PRMKEY,
 AUTHOR PRMKEY,
 BOOKKEY PRMKEY,
 B1 COMPUTED BY ((select a.auname from tauthor a
   where a.author=tbook_author.author)),
 B2 COMPUTED BY ((select a.booknm from tbook a
   where a.unikey=tbook_author.bookkey)),
 CONSTRAINT pk_TBOOK_AUTHOR PRIMARY KEY (UNIKEY)
);
```

Создается таблица с первичным ключом на уровне столбца и уникальным ключом на уровне таблицы и явным указанием символьного набора (character set) и схемы упорядочения (collate).

```
CREATE TABLE TREADER (
 UNIKEY PRMKEY PRIMARY KEY,
 RDNUMB VARCHAR (25)
  character set WIN1251 collate WIN1251,
 RDNAME VARCHAR (60)
  character set WIN1251 NOT NULL  collate PXW_CYRL,
 CONSTRAINT TREADER_RDNAME UNIQUE (RDNAME)
);
```

CREATE TRIGGER

ОПИСАНИЕ

CREATE TRIGGER определяет новый триггер в базе данных. Триггер - отдельная программа, связанная с таблицей или обзором, которая автоматически исполняется, когда строка в таблице или обзоре вставляется, модифицируется или удаляется.

Триггер никогда не называется непосредственно. Вместо этого, когда приложение или пользователь пытаются ВСТАВЛЯТЬ, МОДИФИЦИРОВАТЬ или УДАЛЯТЬ строку в таблице, выполняются все триггеры, связанные с этой таблицей и операцией. Триггеры, определенные для МОДИФИКАЦИИ, на необновляемых обзорах вызываются, даже если никакая модификация не происходит.

Триггер состоит из заголовка и тела.

Заголовок триггера содержит:

- имя триггера, уникальное в пределах базы данных, которая отличает данный триггер от всех других;
- имя таблицы, идентифицирующее таблицу, к которой присоединяется триггер;
- операцию, при которой включается триггер.

Тело триггера содержит:

- необязательный список локальных переменных с указанием их типов данных;
- блок команд на языке процедур и триггеров InterBase в скобках BEGIN - END. Эти команды выполняются при инициации триггера. Блок может самостоятельно включать другие блоки, так что они могут иметь много уровней вложения.

Поскольку каждая команда в теле триггера должна быть закончена точкой с запятой, для пометки конца тела триггера необходимо задать специальный символ ограничитель. Для этого в ISQL используется ко-

манда SET TERM непосредственно перед выдачей CREATE TRIGGER, чтобы определить признак конца, отличный от ";" (точки с запятой). После тела триггера снова помещается команда SET TERM, чтобы восстановить ограничитель ";" (точку с запятой). Если триггер создается из приложения или какой-либо специальной утилиты, то в команде SET TERM нет необходимости, более того, ее появление будет трактоваться, как синтаксическая ошибка.

Триггер связан с таблицей. Владелец таблицы или другой пользователь, имеющий привилегии на таблицу, автоматически получает права выполнять триггер.

Триггеру могут быть предоставлены привилегии на таблицы (в процессе своего выполнения триггер может работать со многими таблицами), так же, как пользователю или процедуре. Для этого используется команда GRANT, но вместо использования конструкции TO username следует указывать TO TRIGGER trigger_name. Привилегии триггера могут отменяться аналогично на основе конструкции REVOKE.

Когда пользователь исполняет действие, которое запускает триггер, он будет иметь привилегии на выполнение, если истинно одно из следующих условий:

- триггер имеет привилегии для доступа и/или изменения данных;
- пользователь имеет привилегии для доступа и/или изменения данных.

Язык хранимых процедур и триггеров InterBase является полным языком программирования. Он включает:

- SQL-команды манипулирования данными (INSERT, UPDATE, DELETE и singleton SELECT);
- SQL-команды и выражения, включая пользовательские функции (UDF - User Defined Functions), подключенные к базе, и генераторы;
- Расширения SQL, включая команды присвоения, управления последовательностью обработки, контекстные переменные (для триггеров), команды инициации событий, исключений и обработки ошибок.

В следующей таблице приведены команды расширения SQL.

Таблица А.23. *Синтаксические конструкции расширения SQL для триггеров*

Команда	***Описание***
BEGIN . . . END	Определяет блок команд рассматриваемых синтаксически как одна команда. Ключевое слово

Команда	*Описание*
	BEGIN помечает начало блока (открывающаяся скобка), ключевое слово END помечает конец блока (закрывающаяся скобка). Ограничитель ";" после BEGIN и END не ставится
Variable = expression	Оператор присвоения; присваивает выражение столбцу, локальной переменной, входному или выходному параметру
/* comment_text */	Комментарий программиста. Может занимать несколько строк
EXCEPTION exception_name	Инициирует указанное исключение. Исключение - это генерируемая пользователем ошибка, которая может быть обработана с помощью оператора WHEN
EXECUTE PROCEDURE proc_name [Var [, var ...]] [RETURNING_VALUES var [, var ...]]	Вызывает на выполнение указанную в proc_name хранимую процедуру со списком входных аргументов, задаваемых после имени процедуры, и списком возвращаемых значений, задаваемого после ключевых слов RETURNING_VALUES. Входные и выходные аргументы должны быть переменными, объявленными в процедуре. Допустимы вложенные процедуры и рекурсия
FOR <select_statement> DO <compound_statement>	Обеспечивает повторение составного оператора, записанного после ключевого слова DO, для каждой из строк, полученных при выполнении запроса, заданного в конструкции <select_statement>. <select_statement> - обычная команда SELECT, за исключением того, что требует конструкции INTO, завершающей команду
<compound_statement>	Составной оператор - это либо отдельный оператор на языке хранимых процедур и триггеров, либо блок, заключенный в скобки BEGIN и END
IF (<condition>) THEN <compound_statement> [ELSE <compound_statement>]	Проверяет условие <condition> и, если оно принимает значение TRUE (истина), выполняет составной оператор, следующий за ключевым словом THEN, иначе выполняет составной оператор, следующий за ключевым словом ELSE, если оно указано. Условие <condition> представляет собой булевское выражение, принимающее одно из значений: (TRUE, FALSE или UNKNOWN). Обычно условие - это пара выражений, соединенных операцией сравнения (=, !=, <, <=…)

Команда	*Описание*
POST_EVENT event_name	Инициирует событие event_name
SUSPEND	В SELECT-процедуре SUSPEND возвращает вызывающему приложению выходные параметры для каждой строки. Не рекомендуется для выполняемых процедур (т.е. вызываемых по EXECUTE PROCEDURE)
WHILE (<condition>) DO <compound_statement>	До тех пор пока условие <condition> является истиной (TRUE), выполняется составной оператор <compound_statement>. Сначала проверяется условие <condition> и, если оно истинно, то выполняется оператор <compound_statement>. Цикл повторяется, пока условие остается истинным
WHEN {<error> [, <error> .../ | ANY} DO <compound_statement>	Команда управления обработкой ошибок. Когда возникает ошибка одного из указанных видов, выполняется составной оператор <compound_statement>. Команда WHEN, если она необходима, должна располагаться в конце блока непосредственно перед END. <error>: EXCEPTION exception_name; SQLCODE errcode или GDSCODE number. ANY указывает на необходимость обработки ошибок любых типов.

СИНТАКСИС

```
CREATE TRIGGER name FOR table
[ACTIVE / INACTIVE]
        {BEFORE / AFTER}
        {DELETE / INSERT / UPDATE}
        [POSITION number]
       AS <trigger_body> terminator

<trigger_body>::=
        [<variable_declaration_list>] <block>

<variable_declaration_list>::=
       DECLARE VARIABLE variable <datatype>;
        [<variable_declaration_list>]

<block>::=
BEGIN
 <compounds>
```

END

```
<compounds>::=<compound_statement> [<compounds>]

<compound_statement>::= {<block> / statement;}

<datatype>::= {
{SMALLINT / INTEGER / FLOAT / DOUBLE PRECISION}
      / {DECIMAL / NUMERIC} [(precision [, scale])]
      / {DATE / TIME / TIMESTAMP}
      / {CHAR / CHARACTER / CHARACTER VARYING / VARCHAR}
            [(1...32767)] [CHARACTER SET charname]
      / {NCHAR / NATIONAL CHARACTER / NATIONAL CHAR}
            [VARYING] [(1...32767)]}
```

Замечание. Типы данных TIME j TIMESTAMP допустимы только в версиях, начиная с 6.

Таблица А.24. *Синтаксические конструкции команды CREATE TRIGGER*

Аргумент	*Описание*
Name	Имя триггера. Имя должно быть уникально в базе данных
Table	Имя таблицы или обзора, при выполнении операций над которыми запускается триггер
ACTIVE	Режим, означающий, что триггер включен (по умолчанию)
INACTIVE	Режим, означающий, что триггер выключен
BEFORE	Указывает, что триггер запускается перед выполнением данной операции
AFTER	Указывает, что триггер запускается после выполнением данной операции
DELETE | INSERT | UPDATE	Указывает операцию, инициирующую триггер

Аргумент	*Описание*
POSITION number	Определяет порядок запуска триггеров для выполнения до или после данной операции. Параметр number - целое в диапазоне от 0 до 32767 (по умолчанию - 0). Триггеры запускаются в порядке возрастания их номеров. Порядок выполнения триггеров на одну и ту же операцию с одинаковыми номерами не определен (случаен)
DECLARE VARIABLE var <datatype>	Объявляет локальные переменные, используемые только в данном триггере. Каждое объявление должно начинаться с DECLARE VARIABLE и заканчиваться ";". var - имя локальной переменной, уникальное в перечне переменных триггера; <datatype> - любой допустимый в InterBase тип
statement	Любой простой оператор, допустимый в SQL для хранимых процедур и триггеров. Каждый оператор (кроме блоков BEGIN - END) должен **заканчиваться ";"**
terminator	Ограничитель, заданный командой SET TERM, указывающий конец тела процедуры. Используется только в ISQL

ПРИМЕР

Триггер ITBOOKAUTHOR_1 обеспечивает формирование при необходимости уникального первичного ключа и проверяет заполнение обязательных полей. Кроме того, триггер проверяет ссылочную целостность по таблицам TAUTHOR и TBOOK. Последнюю проверку можно также реализовать с помощью внешних ключей, что в большинстве случаев и проще и эффективней. Задание POSITION 0 обеспечивает, что данный триггер будет выполняться первым в группе триггеров вставки.

```
SET TERM !! ;
CREATE TRIGGER I_TBOOK_AUTHOR_1 FOR TBOOK_AUTHOR ACTIVE
BEFORE INSERT POSITION 0
as
begin
```

```
  if (new.UNIKEY is NULL) then
new.UNIKEY=GEN_ID(sysnumber,1);
  if (new.AUTHOR is NULL) then exception NO_AUTHORKEY;
  if (new.BOOKKEY is NULL) then exception NO_BOOKKEY;
  if (NOT EXISTS(select * from TAUTHOR where
(AUTHOR=new.AUTHOR)))
   then exception NO_AUTHORKEY;
  if (NOT EXISTS(select * from TBOOK where
(UNIKEY=new.BOOKKEY)))
   then exception NO_BOOKKEY;
end !!
SET TERM ; !!
```

Триггер I_TBOOK_READER_1 используется для генерации уникального номера в таблицу выдачи книг и проверяет правильность заполнения ссылок на таблицы читателей TREADER и книг TBOOK. Последнюю проверку можно также реализовать с помощью внешних ключей. И, наконец, проверяется наличие требуемой книги (поле num_presence таблицы TBOOK). В случае если такая книга есть, счетчик наличия num_presence в таблице TBOOK уменьшается на 1.

```
SET TERM ! ! ;
CREATE TRIGGER I_TBOOK_READER_1 FOR TBOOK_READER ACTIVE
BEFORE INSERT POSITION 0
as
 declare variable num smallint;
begin
  if (new.UNIKEY is NULL) then
   new.UNIKEY=GEN_ID(sysnumber,1);
  if (new.READER is NULL) then exception NO_READERKEY;
  if (new.BOOKKEY is NULL) then exception NO_BOOKKEY;
  if (NOT EXISTS(select * from TREADER
       where (UNIKEY=new.READER)))
   then exception NO_READERKEY;
  if (NOT EXISTS(select * from TBOOK
       where (UNIKEY=new.BOOKKEY)))
   then exception NO_BOOKKEY;
  select num_presence from TBOOK
   where (UNIKEY=new.BOOKKEY) into :num;
  if(num < 1) then exception NO_PRESENCE;
  update TBOOK set num_presence=:num;
end ! !
SET TERM ; !!
```

Триггер D_TBOOK_READER используется для обработки возврата книги, которому соответствует удаление записи из таблицы выдачи книг TBOOK_READER. В результате его работы увеличивается на 1 счетчик

наличия книг num_presence в таблице TBOOK. Вообще говоря, изменение данных в другой таблице порождает включение соответствующих триггеров этой таблицы.

```
SET TERM !! ;
CREATE TRIGGER D_TBOOK_READER FOR TBOOK_READER ACTIVE
BEFORE DELETE POSITION 0
AS
BEGIN
  update tbook set num_presence=num_presence+1
  where unikey=old.bookkey;
END
SET TERM ; !!
```

CREATE VIEW

ОПИСАНИЕ

CREATE VIEW описывает обзор данных, основанных на одной или нескольких таблицах базы данных. Строки обзора определяются командой SELECT, которая задает состав выводимых данных. В базе сохраняется только описание обзора, результаты выборки не запоминаются. В дальнейшем доступ к данным обзора осуществляется как к обычным таблицам, хотя сами они представляют лишь результаты выборки, которая осуществляется в момент обращения к обзору.

Пользователь, создавший обзор, является его владельцем и имеет на него все привилегии, включая право предоставить (GRANT) привилегии другим пользователям, триггерам и хранимым процедурам. Пользователь может иметь привилегии на обзор без того, чтобы иметь доступ к его основным таблицам. При создании обзор только для чтения требует привилегий SELECT для всех основных таблиц, используемых обзором, а обзор, допускающий обновление, требует всех привилегий к основным таблицам.

Указание имен столбцов в обзоре гарантирует, что обзор всегда содержит те же самые столбцы и столбцы всегда имеют те же самые имена. Имена столбцов и их порядок в обзоре соответствуют перечисленным в <select> (даже если перечень столбцов не указан явно).

В перечне столбцов обзора могут содержаться столбцы, основанные на выражениях. Выражение должно возвращать одиночное значение и не может возвращать массив или элемент массива. Если обзор должен включать выражение, то список столбцов должен быть задан явно.

Любые столбцы и используемые в выражении значения должны существовать прежде, чем выражение будет определено.

Предложение (конструкция) SELECT-команды не может включать предложение ORDER BY.

Когда используется конструкция SELECT *, а не список столбцов, порядок вывода основан на порядке, в котором столбцы хранятся в основной таблице.

Конструкция WITH CHECK OPTION дает возможность InterBase предотвратить добавление или модификацию обзора, если он не удовлетворяет условию выборки строк (опция where конструкции select). Не следует использовать WITH CHECK OPTION для обзора только для чтения.

Обзор может быть использовано для обновления (обновляем), если:

- он является подмножеством одиночной таблицы или другого обновляемого обзора;
- все основные столбцы таблицы, исключенные из определения обзора, допускают значения NULL;
- конструкция SELECT обзора не содержит подзапросов, предикатов DISTINCT, предложения HAVING, агрегатных функций, присоединенных таблиц, определяемых пользователем функций (UDF), хранимых процедур.

Если определение обзора не удовлетворяет этим условиям, то это обзор только для чтения (read-only).

Тем не менее, обзоры только для чтения могут быть модифицированы с использованием комбинации определяемых пользователем ограничений логической целостности, триггеров и уникальных индексов.

СИНТАКСИС

CREATE VIEW name *[(LIST_view_col)*
AS <select> *[**WITH CHECK OPTION**]*;

Таблица А.25. *Синтаксические конструкции команды CREATE VIEW*

Аргумент	***Описание***
name	Имя обзора. Должно быть уникальным в базе данных среди имен обзоров, таблиц и процедур
view_col	Имя столбца обзора. Должно быть уникальным в обзоре среди имен столбцов. Обязательно, если обзор включает вычисляемые столбцы, иначе - не обязательно. По умолчанию имена столбцов берутся из соответствующих таблиц
<select>	Задает состав и условия выборки данных
WITH CHECK OPTION	Предотвращает операции INSERT или UPDATE в обновляемом обзоре, если INSERT или UPDATE нарушают условия поиска, указанные в опции WHERE конструкции <seiect> обзора

ПРИМЕР

Обновляемый обзор - перечень книг. Строки таблицы TBOOK, соответствующие рубрикам отсечены. Данный обзор непосредственно не пригоден для обновления, поскольку содержит поля из нескольких таблиц.

```
CREATE VIEW NORUBRICS (
    UNIKEY,
    BOOKNM,
    B1)
AS  select a.UNIKEY, a.BOOKNM, b.B1 from tbook a,
tbook_author b  where a.matherkey>0 and a.unikey=b.bookkey;
```

Следующий обзор получен из предыдущего удалением столбца B1. Теперь все столбцы обзора выбираются из одной таблицы. Обзор можно обновлять. Добавить здесь конструкцию WITH CHECK OPTION нельзя, поскольку контроль осуществляется построчно, а столбца matherkey в обзоре нет,

```
CREATE VIEW NORUBRICS (
    UNIKEY,
    BOOKNM)
AS  select a.UNIKEY, a.BOOKNM from tbook a
 where a.matherkey>0;
```

Команда обновления может выглядеть так:

```
update norubric1 set
BOOKNM='Тайна за 7 печатями и одной пломбой'
whereunikey=18;
```

Теперь рассмотрим еще раз первый обзор и добавим к нему триггер. После этого можно будет выполнять команды обновления.

```
CREATE TRIGGER NORUBRICS_BU FOR NORUBRICS ACTIVE
BEFORE UPDATE POSITION 0
AS
BEGIN
  if((new.unikey is NULL or new.unikey=old.unikey)
   and (new.b1 is NULL or new.bl=old.bl)) then
    update tbook set tbook.BOOKNM=new.booknm
    where tbook.unikey=old.unikey;
END
```

Отметим, что команды удаление и вставка будут отклонены. Для их подключения необходимо добавить соответствующие триггеры.

DECLARE CURSOR

ОПИСАНИЕ

DECLARE CURSOR объявляет набор строк, которые могут быть получены на его основе. DECLARE CURSOR - первая команда группы команд табличных курсоров, которые должны использоваться последовательно.

Конструкция выборки <select> определяет предложение SELECT, задающее выбираемые строки. Предложение SELECT не должно содержать конструкции INTO и ORDER BY .

Конструкция FOR UPDATE OF необходима для обновления и удаления строк в режиме, использующем конструкцию WHERE CURRENT OF в операциях обновления (UPDATE) и удаления (DELETE).

Курсор представляет собой односторонний указатель в упорядоченном наборе строк, полученных по выражению выбора команды DECLARE CURSOR. Это предполагает последовательный доступ к полученным строкам. Имеются четыре связанных команды курсора.

Таблица А.26. *Команды работы с курсором*

№ п/п	*Команда*	*Назначение*
1	DECLARE CURSOR	Объявляет курсор. Предложение SELECT задает получаемые курсором строки
2	OPEN	Получает строки, специфицированные в DECLARE CURSOR. Полученные строки образуют активный набор курсора
3	FETCH	Получает текущую строку активного набора, начиная с первой. Последовательное выполнение команд FETCH перемещает курсор по набору
4	CLOSE	Закрывает курсор и освобождает системные ресурсы

Команда может использоваться в SQL и DSQL.
SQL форма:

```
DECLARE cursor CURSOR FOR <select>
  [FOR UPDATE OF LIST_<col> ] ;
```

DSQL форма:

DECLARE cursor CURSOR FOR <statement_id>

BLOB форма:

DECLARE cursor **(BLOB)**

Таблица А.27. *Синтаксические конструкции команды DECLARE CURSOR*

Аргумент	*Описание*
cursor	Имя курсора
<select>	Определяет состав получаемых строк
FOR UPDATE OF *LIST_*<col>	Разрешает команды UPDATE и DELETE для указанных столбцов полученных строк
<statement_id>	Имя ранее подготовленной команды SQL. В данном случае команды SELECT. Только в DSQL.

ПРИМЕР

Следующая команда внедренного SQL объявляет курсор с условием поиска:

```
EXEC SQL
DECLARE Mcursor CURSOR FOR
SELECT UNIKEY, MATHERKEY, BOOKNM FROM TBOOK;
```

Следующая команда DSQL объявляет курсор для предварительно подготовленной команды Dquery:

```
DECLARE Dcursor CURSOR FOR Dquery;
```

DECLARE CURSOR (BLOB)

ОПИСАНИЕ

DECLARE CURSOR (BLOB) объявляет курсор для чтения и вставки BLOB-данных. Курсор BLOB может быть связан только с одним столбцом BLOB.

Для чтения сегментов BLOB в случаях, когда переменная базового языка меньше длины сегмента BLOB, следует объявить курсор BLOB с опцией MAXIMUM_SEGMENT. Если длина меньше сегмента BLOB, FETCH вернет заданное число байтов, иначе - весь сегмент (принято по умолчанию).

Эта команда может быть использована в SQL.

СИНТАКСИС

```
DECLARE cursor CURSOR FOR
{READ BLOB column FROM table
                / INSERT BLOB column INTO table}
                [FILTER [FROM subtype/ TO subtype7
                [MAXIMUM_SEGMENT length];
```

Таблица А.28. *Синтаксические конструкции команды DECLARE CURSOR (BLOB)*

Аргумент	*Описание*
cursor	Имя курсора BLOB
column	Имя столбца BLOB
table	Имя таблицы
READ BLOB	Объявляет операцию чтения BLOB
INSERT BLOB	Объявляет операцию записи BLOB
/FILTER /"FROM subtype/ TO subtype]	Специфицирует необязательный фильтр BLOB, используемый для перевода BLOB из одного пользовательского формата (типа) в другой. Тип определяет, какие фильтры используются для перевода
MAXIMUM_SEGMENT length	Длина локальной переменной для получения данных BLOB командой FETCH

ПРИМЕР

Следующая команда внедренного SQL объявляет курсор BLOB для чтения и использует опцию MAXIMUM_SEGMENT:

```
EXEC SQL
       DECLARE BC CURSOR FOR
       READ BLOB JOB_REQUIREMENT FROM JOB MAXIMUM_SEGMENT
40;
```

Команда внедренного SQL объявляет курсор BLOB для вставки:

```
EXEC SQL
DECLARE BlobCur CURSOR FOR
       INSERT BLOB REFERAT INTO TBOOK;
```

DECLARE EXTERNAL FUNCTION

ОПИСАНИЕ

DECLARE EXTERNAL FUNCTION создает описание в базе пользовательской функции (UDF - User Defined Function), включая ее имя, входные параметры, если они требуются, возвращаемое значение. Сама функция размещается в библиотеке вне базы данных.

Функция, вызываемая из базы, должна быть откомпилирована в соответствии с CDECL соглашением.

Опция 'entryname' - имя, под которым функция хранится в библиотеке. Имя функции в библиотеке и имя, объявленное в базе, вообще говоря, различны.

Нельзя использовать DECLARE EXTERNAL FUNCTION, если база размещена на сервере NetWare. Сервер NetWare не поддерживает библиотеки UDF.

СИНТАКСИС

```
DECLARE EXTERNAL FUNCTION name [LIST_<datatypes>
RETURNS {<datatype> [BY VALUE] | CSTRING (int)} [FREE_IT]
ENTRY_POINT 'entryname' MODULE_NAME 'modulename';

<datatypes> ::= <datatype> / CSTRING (int)
```

Таблица А.29. *Синтаксические конструкции команды DECLARE EXTERNAL FUNCTION*

Аргумент	*Описание*
name	Имя UDF
<datatype>	Тип данных входных и выходного параметра. Все входные параметры передаются в UDF по ссылке. Возврат может осуществляться и по значению
RETURNS	Специфицирует возвращаемое функцией значение
BY VALUE	Указывает, что возврат осуществляется по значению, иначе - по ссылке
CSTRING(int)	Указывает, что UDF возвращает заканчивающуюся 0 строку длины int байтов
FREE_IT	Признак того, что для возвращаемого значения в UDF была явно выделена память и InterBase должен ее освободить. В C, C++ для выделения памяти следует использовать функцию malloc

Аргумент	*Описание*
'entryname'	Строка в апострофах, специфицирующая имя функции в библиотеке
'modulename'	Строка в апострофах, специфицирующая имя файла (библиотеки), в котором размещены UDF. В Widows библиотека должна быть размещена в поддиректории UDF директорий установки InterBase для версии 6. Для всех версий нужную директорию можно определить по расположению DLL ib_udf.dll.

ПРИМЕР

Объявляется функция IFC , возвращающую символьную строку и получающую 3 параметра.

```
/* IFC D,C1,C2)=(D>0)?C1:C2 */
DECLARE EXTERNAL FUNCTION IFC
    DOUBLE PRECISION,
    VARCHAR  (256),
    VARCHAR (256)
    RETURNS VARCHAR (256)
    ENTRY_POINT '_if_c' MODULE_NAME 'MyDLL.dll';
```

DECLARE FILTER

ОПИСАНИЕ

DECLARE FILTER задает информацию о существующем фильтре BLOB в базе данных: где он находится, его имя и подтип BLOB, с которым он работает. Фильтр BLOB - это пользовательская программа, которая преобразовывает данные, сохраненные в столбцах BLOB из одного подтипа в другой.

INPUT_TYPE и OUTPUT_TYPE вместе определяют поведение фильтра BLOB. Каждый фильтр, объявленный в базе данных должен иметь уникальную комбинацию INPUTTYPE и OUTPUT_TYPE (целочисленные коды). InterBase обеспечивает встроенный тип 1, для обработки текста. Определяемые пользователем типы должны быть заданы как отрицательные значения.

Параметр 'entryname' задает имя фильтра BLOB в библиотеке. Когда приложение использует фильтр BLOB, это вызывает функцию фильтра с указанным именем.

Нельзя использовать DECLARE FILTER при создании базы данных на сервере NetWare. Фильтры BLOB не поддерживаются на серверах NetWare.

СИНТАКСИС

```
DECLARE FILTER filter
INPUT_TYPE subtype OUTPUT_TYPE subtype
      ENTRY_POINT 'entryname' MODULE_NAME 'modulename';
```

Таблица А.30. *Синтаксические конструкции команды DECLARE FILTER*

Аргумент	*Описание*
filter	Имя фильтра. Должно быть уникальным среди имен фильтров в базе данных
INPUT_TYPE subtype	Определяет подтип BLOB, данные из которого должны быть преобразованы
OUTPUT_TYPE subtype	Определяет подтип BLOB, в который должны быть преобразованы данные
'entryname'	Строка в апострофах, определяющая имя фильтра BLOB, как он хранится в библиотеке, указанной в 'modulename'
'modulename'	Строка в апострофах, специфицирующая имя файла (библиотеки), в котором размещен фильтр. В Widows библиотека должна быть размещена в поддиректории UDF директорий установки InterBase для версии 6. Для всех версий нужную директорию можно определить по расположению DLL ib_udf.dll.

ПРИМЕР

Следующая инструкция объявляет фильтр BLOB:

```
DECLARE FILTER TFILTER
INPUT_TYPE -1 OUTPUT_TYPE -2
ENTRY_POINT '_TFilter'
MODULE_NAME 'MYDLL.dll';
```

DESCRIBE

ОПИСАНИЕ

Команда DESCRIBE используется в двух случаях.

Во-первых, при описании команды вывода DESCRIBE записывает в XSQLDA описание столбцов из списка выбора предварительно подготовленной команды. Если команда PREPARE включала предложение INTO, нет необходимости использовать DESCRIBE как команду вывода.

Во-вторых, при описании команды ввода DESCRIBE записывает в XSQLDA описание динамических параметров, указанных в предварительно подготовленной команде.

DESCRIBE является одной из команд, образующих группу команд DSQL.

Таблица А.31. *Команды, работающие с XSQLDA*

Команда	*Назначение*
PREPARE	Подготавливает к выполнению команду DSQL
DESCRIBE	Заносит в XSQLDA данные команды
EXECUTE	Выполняет ранее подготовленную команду
EXECUTEIMMEDIATE	Подготавливает команду DSQL, выполняет ее один раз и очищает ее

Для операций ввода и вывода должны выдаваться отдельные команды DESCRIBE. Чтобы сохранить информацию о динамическом параметре, должно использоваться ключевое слово INPUT.

Когда DESCRIBE используется для вывода, то, если значение, возвращенное в sqld-поле в XSQLDA, больше, чем sqln-поле, необходимо:

- выделить большее количество памяти для структур XSQLVAR,
- снова выдать команду DESCRIBE.

Та же самая структура XSQLDA может при желании использоваться как для ввода, так и для вывода.

Команда DESCRIBE может использоваться в SQL.

СИНТАКСИС

DESCRIBE [**OUTPUT** / **INPUT**] statement
{**INTO** / USING} **SQL DESCRIPTOR** xsqlda;

Таблица А.32. *Синтаксические конструкции команды DESCRIBE*

Аргумент	*Описание*
OUTPUT	Указывает, что информация о столбце должна быть возвращена в XSQLDA (значении по умолчанию)
INPUT	Указывает, что информация динамического параметра должна быть сохранена в XSQLDA

Аргумент	*Описание*
Statement	Ранее определенный для команды DESCRIBE псевдоним. Псевдонимы определяются в команде PREPARE
{INTO | USING,/ SQL DESCRIPTOR xsqlda	Определяет XSQLDA для команды DESCRIBE

Следующая внедренная команда SQL получает информацию о выводе команды SELECT:

```
EXEC SQL
      DESCRIBE Q1 INTO Work_xsqlda;
```

Следующая внедренная команда SQL сохраняет информацию о переданных динамических параметрах для команды, которая будет затем выполнена:

```
EXEC SQL
      DESCRIBE INPUT Q2 USING SQL DESCRIPTOR Work_xsqlda;
```

DISCONNECT

ОПИСАНИЕ

DISCONNECT закрывает указанную базу данных, идентифицированную дескриптором базы, или все базы данных, освобождает ресурсы, используемые подключенной базой, обнуляет дескрипторы базы, записывает данные транзакции по умолчанию, если не включена управляющая опция GPRE, и возвращает ошибку, если какая-либо транзакция, отличная от умалчиваемой, не завершена. Прежде чем выполнить DISCONNECT, следует записать или откатить все транзакции в базе, соединение с которой разрывается.

Чтобы вновь соединиться с базой данных, закрытой DISCONNECT, ледует выдать команду CONNECT.

Эта команда может использоваться в SQL.

СИНТАКСИС

DISCONNECT *{{***ALL** / **DEFAULT***} | LIST*_dbhandle*};*

Таблица А.33. *Синтаксические конструкции команды DISCONNECT*

Аргумент	*Описание*
ALL| DEFAULT	Любое ключевое слово закрывает все открытые базы данных
dbhandle	Ранее объявленный дескриптор, идентифицирующий отсоединяемую базу данных

Следующие команды внедренного SQL закрывают все базы данных:

```
EXEC SQL
      DISCONNECTDEFAULT;
EXEC SQL
      DISCONNECTALL;
```

Следующие команды внедренного **SQL закрывают** базы данных, идентифицированные их дескрипторами:

```
EXEC SQL
      DISCONNECTDB1;
EXEC SQL
      DISCONNECT DB1, DB2;
```

DROP DATABASE

ОПИСАНИЕ

DROP DATABASE удаляет подключенную базу данных, включая все связанные вторичные, теневые базы и журналы. Удаляются все данные, которые она содержит. База данных может быть удалена ее создателем и пользователем SYSDBA.

СИНТАКСИС

```
DROP DATABASE;
```

ПРИМЕР

```
DROP D_Base;
```

DROP DOMAIN

ОПИСАНИЕ

DROP DOMAIN удаляет существующее определение домена из базы данных. Если домен используется в каком-либо определении столбца в базе данных, DROP не выполняется. Для предотвращения отказа следует

перед выполнением DROP DOMAIN выполнить команды ALTER TABLE, чтобы удалить столбцы, основанные на домене. Домен может быть удален его создателем и пользователем SYSDBA.

СИНТАКСИС

```
DROP DOMAIN name;
```

Таблица А.34. *Синтаксические конструкции команды DROP DOMAIN*

Аргумент	*Описание*
name	Имя существующего домена

ПРИМЕР

```
DROP DOMAIN DMONTH;
```

DROP EXCEPTION

ОПИСАНИЕ

DROP EXCEPTION удаляет исключение из базы данных. Исключения, используемые в существующих процедурах или триггерах, не могут быть удалены. Отображение списка исключений и использующих их процедур и триггеров можно получить с помощью команды SHOW EXCEPTION. Исключение может быть удалено его создателем и пользователем SYSDBA.

СИНТАКСИС

```
DROP EXCEPTION name;
```

Таблица А.35. *Синтаксические конструкции команды DROP EXCEPTION*

Аргумент	*Описание*
name	Имя существующего сообщения исключения

ПРИМЕР

```
DROP EXCEPTION NO_AUTHORKEY;
```

DROP EXTERNAL FUNCTION

ОПИСАНИЕ

DROP EXTERNAL FUNCTION удаляет UDF-объявление (объявление функции пользователя) из базы данных. Удаление UDF-объявления из базы данных не удаляет саму функцию из связанной UDF-библиотеки, но делает ее недоступной в базе данных. Если объявление удалено, любые приложения, которые используют UDF, возвратят ошибки во время выполнения.

Объявление UDF может быть удалено его создателем и пользователем SYSDBA.

UDF не доступны на серверах NetWare. Если UDF все же объявлена, для удаления объявления следует выполнить команду DROP EXTERNAL FUNCTION.

СИНТАКСИС

```
DROP EXTERNAL FUNCTION name;
```

Таблица А.36. *Синтаксические конструкции команды DROP EXTERNAL FUNCTION*

Аргумент	*Описание*
name	Имя существующей UDF

ПРИМЕР

```
DROP EXTERNAL FUNCTION if;
```

DROP FILTER

ОПИСАНИЕ

DROP FILTER удаляет объявление фильтра BLOB из базы данных. Удаление объявления фильтра BLOB из базы не удаляет сам фильтр BLOB из связанной библиотеки, но делает его недоступным в базе данных. Если объявление удалено, любые приложения, которые используют фильтр, возвратят ошибки во время выполнения.

DROP FILTER завершится аварийно, если он используется каким-либо объектом базы, например процедурой или триггером.

Объявление фильтра может быть удалено его создателем и пользователем SYSDBA.

Фильтры BLOB не доступны на серверах NetWare. Если фильтр все же объявлен, для удаления объявления следует выполнить команду DROP FILTER.

СИНТАКСИС

DROP FILTER name;

Таблица А.37. *Синтаксические конструкции команды DROP FILTER*

Аргумент	*Описание*
name	Имя существующего фильтра

ПРИМЕР

```
DROP FILTER TFILTER;
```

DROP INDEX

ОПИСАНИЕ

DROP INDEX удаляет пользовательский индекс из базы. Команда неприменима для системных индексов, обеспечивающих логическую целостность данных, таких как UNIQUE, PRIMARY KEY, FOREIGN KEY.

Индекс может быть удален только после окончания его использования всеми приложениями и только создателем и пользователем SYSDBA

СИНТАКСИС

DROP INDEX name;

Таблица А.38. *Синтаксические конструкции команды DROP INDEX*

Аргумент	*Описание*
name	Имя существующего индекса

ПРИМЕР

```
DROP INDEX TREADER_RDNAME;
```

DROP PROCEDURE

ОПИСАНИЕ

DROP PROCEDURE удаляет существующую хранимую процедуру из базы. Процедура, используемая другими процедурами, триггерами или обзорами, не может быть удалена. Она также не может быть удалена, пока не завершится использующее ее приложение (транзакция).

Отображение списка процедур и их связей с другими процедурами, триггерами, исключениями, таблицами можно получить с помощью команды SHOW PROCEDURE.

Хранимая процедура может быть удалена ее создателем и пользователем SYSDBA.

СИНТАКСИС

```
DROP PROCEDURE name;
```

Таблица А.39. *Синтаксические конструкции команды DROP PROCEDURE*

Аргумент	*Описание*
name	Имя существующей хранимой процедуры

ПРИМЕР

```
DROP PROCEDURE PBOOKAUTHOR;
```

DROP SHADOW

ОПИСАНИЕ

DROP SHADOW удаляет набор файлов тени и отключает процесс ведения тени. Для просмотра набора файлов тени можно использовать команду SHOW DATABASE.

Тень может быть удалена ее создателем и пользователем SYSDBA.

СИНТАКСИС

```
DROP SHADOW set_num;
```

Таблица А.40. *Синтаксические конструкции команды DROP SHADOW*

Аргумент	*Описание*
set_num	Положительное целое, идентифицирующее набор файлов тени

ПРИМЕР

```
DROP SHADOW 1;
```

DROP TABLE

ОПИСАНИЕ

DROP TABLE удаляет из таблицы данные, а из базы - описание самой таблицы, индексы и все, связанные с таблицей триггеры. Не могут быть удалены таблицы, которые используются в SQL-выражениях, обзорах, ограничениях логической целостности, хранимых процедурах. Также

нельзя удалить таблицу до тех пор, пока она используется в незавершенной транзакции.

При выдаче команды DROP TABLE для внешней таблицы происходит удаление только ее описания из базы. Внешний файл не удаляется.

Таблица может быть удалена ее создателем и пользователем SYSDBA.

СИНТАКСИС

DROP TABLE name;

Таблица А.41. *Синтаксические конструкции команды DROP TABLE*

Аргумент	*Описание*
name	Имя существующей таблицы

ПРИМЕР

```
DROP TABLE tbook;
```

DROP TRIGGER

ОПИСАНИЕ

DROP TRIGGER удаляет триггер из базы данных. Системные триггеры, подобные триггерам, создаваемым по ограничениям CHECK, не могут удаляться командой DROP TRIGGER. Для удаления таких триггеров можно использовать команду ALTER TABLE с опцией DROP для соответствующих конструкций CHECK.

Нельзя удалить триггер до тех пор, пока он используется в незавершенной транзакции.

Триггер может быть удален ее создателем и пользователем SYSDBA.

Для временного отключения триггера следует использовать команду ALTER TRIGGER, указав в заголовке триггера INACTIVE.

СИНТАКСИС

```
DROP TRIGGER name;
```

Таблица А.42. *Синтаксические конструкции команды DROP TRIGGER*

Аргумент	*Описание*
name	Имя существующего триггера

ПРИМЕР

```
DROP TRIGGER I_TBOOK_1;
```

DROP VIEW

ОПИСАНИЕ

DROP VIEW удаляет описание обзора из базы, если оно не используется в другом обзоре, хранимой процедуре или ограничении CHECK.

Обзор может быть удален его создателем и пользователем SYSDBA.

СИНТАКСИС

```
DROP VIEW name;
```

Таблица А.43. *Синтаксические конструкции команды DROP VIEW*

Аргумент	*Описание*
name	Имя существующего обзора

ПРИМЕР

```
DROP VIEW NORUBRICS;
```

END DECLARE SECTION

ОПИСАНИЕ

END DECLARE SECTION используется в приложениях с внедренным SQL для идентификации конца объявлений переменных базового языка в отношении переменных, которые будут использоваться в последующих командах SQL.

Эта инструкция может использоваться в SQL.

СИНТАКСИС

```
END DECLARE SECTION;
```

ПРИМЕР

Следующие инструкции внедренного SQL объявляют раздел, переменные базового языка и фиксируют конец объявлений:

```
EXEC SQL
BEGIN DECLARE SECTION;
BASED ON TBOOK.REFERAT blob_id;
BASED ON TBOOK.REFERAT.SEGMENT blob_segment_buf;
BASED ON TBOOK.UNIKEY key;
```

```
unsigned short blob_seg_len;
EXEC SQL
END DECLARE SECTION;
```

EVENT INIT

ОПИСАНИЕ

EVENT INIT задает первый этап в двухэтапном механизме синхронизации событий InterBase:

1. EVENT INIT регистрирует интерес приложения к событию.
2. EVENT WAIT заставляет приложение ожидать появления указанного события.

EVENT INIT регистрирует интерес приложения к событиям; перечень событий задается в виде списка в круглых скобках. Список должен соответствовать событиям, зарегистрированным в базе данных хранимыми процедурами или триггерами. Если приложение регистрирует интерес к нескольким событиям в одном EVENT INIT, то, когда происходит одно из этих событий, приложение должно определить, какое именно событие произошло. События инициируются командой POST_EVENT в хранимой процедуре или триггере.

Диспетчер событий следит за регистрацией интереса к событиям. При записи транзакций, инициировавших события, диспетчер уведомляет об этом заинтересованные приложения.

Эта инструкция может использоваться в SQL.

СИНТАКСИС

EVENT INIT request_name *[*<dbhandle>*]*
(*LIST*_<eventnames>);

<eventnames>::="string" / :<variable>

Таблица А.44. *Синтаксические конструкции команды EVENT INIT*

Аргумент	*Описание*
request_name	Дескриптор события
<dbhandle>	Специфицирует базу данных, в которой контролируется возникновение событий. Если пропущено, используется база данных, для которой была выдана последняя команда SET DATABASE
"<string>"	Уникальное имя, идентифицирующее событие, которое инициируются командой POST_EVENT в хранимой процедуре или триггере

Аргумент	*Описание*
:<variable>	Символьный массив базового языка, содержащий связанное с командой уникальное имя, идентифицирующее событие, которое инициируются командой POST_EVENT в хранимой процедуре или триггере

ПРИМЕР

Следующие команды внедренного SQL регистрируют имя события приложения и указывают, что программа переходит к его ожиданию:

Регистрирует интерес приложения к событиям.

```
EXEC SQL
       EVENT INIT Event_1 MYBASE ("Ev_001");

. . .
```

Ожидается событие.

```
EXEC SQL
       EVENT WAIT Event_1;
```

Само событие устанавливается внутри базы в триггере или хранимой процедуре.

```
if(ABC > 1000)
 begin
  . . .
  POST_EVENT 'Ev_001';
   . . .
 end
 . . .
```

EVENT WAIT

ОПИСАНИЕ

EVENT WAIT задает второй этап в двухэтапном механизме синхронизации событий InterBase.

После того как приложение регистрирует интерес к событию, EVENT WAIT заставляет приложение ожидать появления указанного события.

Эта инструкция может использоваться в SQL.

СИНТАКСИС

```
EVENT WAIT request_name;
```

Таблица А.45. *Синтаксические конструкции команды EVENT WAIT*

Аргумент	*Описание*
request_name	Дескриптор события, объявленный ранее в команде EVENT INIT

ПРИМЕР

Следующие команды внедренного SQL регистрируют имя события приложения и указывают, что программа переходит к его ожиданию:

Регистрирует интерес приложения к событиям.

```
EXEC SQL
       EVENT INIT Event_1 MYBASE ("Ev_001");

. . .
```

Ожидается событие.

```
EXEC SQL
       EVENT WAIT Event_1;
```

Само событие устанавливается внутри базы в триггере или хранимой процедуре.

```
if(ABC > 1000)
 begin
  . . .
  POST_EVENT 'Ev_001';
   . . .
 end
 . . .
```

EXECUTE

ОПИСАНИЕ

EXECUTE выполняет предварительно подготовленную команду DSQL. Это - одна из группы команд для выполнения команд DSQL.

Таблица А.46. *Команды подготовки и выполнения инструкций SQL*

Команда	*Назначение*
PREPARE	Подготавливает команду DSQL к выполнению
DESCRIBE	Заполняет XSQLDA информацией об инструкции

Команда	*Назначение*
EXECUTE	Выполняет ранее подготовленную команду
EXECUTE IMMEDIATE	Подготавливает команду DSQL, выполняет и очищает ее

Прежде чем команда будет выполнена, она должна быть подготовлена командой PREPARE. В качестве исполняемой команды могут выступать любые SQL-команды описания данных, манипуляции с данными или управления транзакциями. Один раз подготовленная команда может быть затем выполнена любое число раз.

В приложениях SQL, выполняющих одновременно несколько транзакций, может использоваться конструкция TRANSACTION для указания, какой именно транзакцией контролируется операция EXECUTE.

Конструкция USING DESCRIPTOR позволяет команде EXECUTE извлекать параметры команд из предварительно загруженной данными приложения структуры XSQLDA. Конструкция USING DESCRIPTOR необходима только при использовании команд, имеющих динамические параметры.

Конструкция INTO DESCRIPTOR позволяет команде EXECUTE сохранять для приложения, возвращаемые выполненной командой значения, в указанной структуре XSQLDA. Требуется только для инструкций DSQL с возвращаемыми значениями.

Если команда EXECUTE предусматривает и конструкцию USING DESCRIPTOR, и конструкцию INTO DESCRIPTOR, то необходимо задавать две структуры XSQLDA.

Эта инструкция может использоваться в SQL.

СИНТАКСИС

```
EXECUTE [TRANSACTION transaction] statement
 [USING SQL DESCRIPTOR xsqlda] [INTO SQL DESCRIPTOR
xsqlda];
```

Таблица А.47. *Синтаксические конструкции команды EXECUTE*

Аргумент	*Описание*
TRANSACTION	
transaction	Специфицирует транзакцию, контролирующую выполнение команды

Аргумент	*Описание*
statement	Псевдоним ранее подготовленного к выполнению предложения
USING SQL DESCRIPTOR	Указывает, что соответствующие значения в подготовленном предложении должны выбираться из специфицированного XSQLDA
INTO SQL DESCRIPTOR	Указывает, что возвращаемые командой значения должны быть сохранены в специфицированной XSQLDA
xsqlda	XSQLDA-переменная базового языка

ПРИМЕР

Следующая команда внедренного SQL выполняет предварительно подготовленную команду DSQL:

```
EXEC SQL
        EXECUTE DSQL_Query;
```

Следующая команда внедренного SQL выполняет предварительно подготовленную команду с параметрами, сохраненными в XSQLDA:

```
EXEC SQL
        EXECUTE DSQL_Query USING DESCRIPTOR Work_xsqlda;
```

Следующая команда внедренного SQL выполняет предварительно подготовленную команду с параметрами в одном XSQLDA и сохраняет результаты в другом XSQLDA:

```
EXEC SQL
        EXECUTE DSQL_Query USING DESCRIPTOR xsqlda_In INTO
DESCRIPTOR xsqlda_Out;
```

EXECUTE IMMEDIATE

ОПИСАНИЕ

EXECUTE IMMEDIATE подготавливает команду DSQL, сохраняет в переменной базового языка или символьной строке, выполняет ее и очищает. Чтобы подготовить команду для многократного использования, вместо EXECUTE IMMEDIATE следует использовать команды PREPARE и EXECUTE.

Конструкция TRANSACTION может применяться в приложениях, использующих одновременно несколько транзакций, чтобы специфици-

ровать, под управлением какой именно из них работает EXECUTE IMMEDIATE.

Команды SQL, предназначенные для выполнения, должны быть записаны либо в переменных базового языка, либо в символьных строках. Они могут содержать любые команды определения данных или манипулирования данными, которые не имеют возвращаемых значений.

Конструкция USING DESCRIPTOR позволяет команде EXECUTE IMMEDIATE извлекать параметры команд из структуры XSQLDA, предварительно загруженной данными приложения. Конструкция USING DESCRIPTOR необходима только при использовании команд, имеющих динамические параметры.

Эта команда может использоваться в SQL.

СИНТАКСИС

```
EXECUTE IMMEDIATE [TRANSACTION transaction]
 {:<variable> / "string"} [USING SQL DESCRIPTOR xsqlda ] ;
```

Таблица А.48. *Синтаксические конструкции команды EXECUTE IMMEDIATE*

Аргумент	*Описание*
TRANSACTION transaction	Специфицирует транзакцию, контролирующую выполнение команды
:<variable>	Имя переменной базового языка, содержащей предназначенную для выполнения команду
USING SQL DESCRIPTOR	Указывает, что соответствующие значения в подготовленном предложении должны выбираться из специфицированного XSQLDA
xsqlda	XSQLDA-переменная базового языка

ПРИМЕР

Следующая команда внедренного SQL готовит и выполняет команды, помещенные в переменной базового языка.

```
EXEC SQL
       EXECUTE IMMEDIATE :Query_Data;
```

EXECUTE PROCEDURE

ОПИСАНИЕ

EXECUTE PROCEDURE вызывает указанную хранимую процедуру.

Процедура, вызываемая из ISQL, по своему смыслу не может возвращать какие-либо значения. Процедура, вызываемая из другой процедуры или триггера, может возвращать значения, также может возвращать значения и процедура, вызываемая из приложения на базовом языке. Соответственно вызов процедуры в этих случаях имеет различный синтаксис.

СИНТАКСИС ДЛЯ ISQL

```
EXECUTE PROCEDURE name [LIST_param];
```

СИНТАКСИС ДЛЯ SQL (ВЫЗОВА ИЗ ПРОЦЕДУР И ТРИГГЕРОВ)

```
EXECUTE PROCEDURE name [LIST_param]
[RETURNING_VALUES [LIST_param]
```

СИНТАКСИС ДЛЯ DSQL

```
EXECUTE PROCEDURE [TRANSACTION transaction]
name [LIST_<dparam>]
[RETURNING_VALUES LIST_<dparam>];

<dparam> ::= :param [[INDICATOR]:indicator]
```

Таблица А.49. *Синтаксические конструкции команды EXECUTE PROCEDURE*

Аргумент	*Описание*
name	Имя существующей хранимой процедуры
param	Входные параметры; должны быть константами для ISQL либо константами или переменными для SQL и DSQL
TRANSACTION transaction	Имя транзакции, в рамках которой выполняется команда
[INDICATOR] :indicator	Имя базовой переменной, в которую записывается признак того, что возвращенное значения - NULL

ПРИМЕР

ISQL вызов. Применяется для процедур, выполняющих **изменение** данных.

```
EXECUTE PROCEDURE ABC 100;
```

Вызов процедуры из другой процедуры, возвращающей значение **в** переменную AUTHORS

```
execute procedure PBUTHOR :UNIKEY
  RETURNING_VALUES :AUTHORS;
```

Аналогичный вызов процедуры из приложения, возвращающей значение в переменную AUTHORS

```
EXEC SQL
EXECUTE PROCEDURE TRANSACTION tr_1
PBUTHOR :wUNIKEY
RETURNING_VALUES :wAUTHORS;
```

FETCH

ОПИСАНИЕ

FETCH выбирает в программу одну строку из активного набора курсора. Первая команда FETCH выбирает первую строку активного набора. Последующие команды FETCH продвигают курсор последовательно по активному набору, пока не будет достигнут его конец, тогда SQLCODE устанавливается в 100.

Курсор - это односторонний указатель в упорядоченном наборе строк, полученных выражением SELECT в команде DECLARE CURSOR. Курсор допускает только последовательный доступу к отысканным строкам.

Имеются четыре связанных команды курсора:

Таблица А.50. *Команды работы с курсором*

№ п/п	*Команда*	*Назначение*
1	DECLARE CURSOR	Объявляет курсор. Конструкция SELECT определяет состав столбцов курсора
2	OPEN	Получает строки, указанные в DECLARE CURSOR. Полученные строки образуют активный набор курсора

№ п/п	*Команда*	*Назначение*
3	FETCH	Получает очередную строку активного набора, начиная с первой. Последовательное выполнение FETCH продвигает курсор по набору
4	CLOSE	Закрывает курсор и освобождает системные ресурсы

Число, размер, тип данных и порядок столбцов в FETCH должен быть тем же, что и в конструкции запроса соответствующей ему команды DECLARE CURSOR. Если это не так, результат может оказаться неверным.

Эта команда может использоваться в SQL и DSQL.

СИНТАКСИС

SQL form:

```
FETCH Cursor
[INTO :hostvar [[INDICATOR] :indvar]
              [, :hostvar [[INDICATOR] :indvar] ...]];
```

DSQL form:

```
FETCH cursor {INTO | USING} SQL DESCRIPTOR xsqlda
```

Blob form:

См. FETCH (BLOB).

Таблица А.51. *Синтаксические конструкции команды FETCH*

Аргумент	*Описание*
Cursor	Имя открытого курсора, используемого для получения строк
:hostvar	Переменная базового языка, получающая данные по команде FETCH, Необязательна, если FETCH получает строки для удаления или обновления. Требуется, если строка выводится перед удалением или обновлением

Аргумент	*Описание*
:indvar	Управляющая переменная, указывающая, что столбец содержит неизвестную величину или NULL
[INTO | USING] SQLDESCRIPTOR	Означает, что значение должно быть возвращено в указанной XSQLDA
xsqlda	Переменная XSQLDA базового языка

ПРИМЕР

```
EXEC SQL
DECLARE Mcursor CURSOR FOR
SELECT UNIKEY, MATHERKEY, BOOKNM FROM TBOOK;

. . .

do
{
 EXEC SQL
       FETCH Mcursor INTO :wUNIKEY, :wMATHERKEY, :wBOOKNM;
 if(SQLCODE)break;
 . . .
}while(1);
```

GRANT

ОПИСАНИЕ

GRANT устанавливает права на объекты базы данных пользователям или другим объектам базы данных. Когда объект создается, права на него имеет только его создатель и только он может выдавать права другим пользователям или объектам.

Для доступа к таблице или обзору пользователь или объект нуждается в правах на SELECT, INSERT, UPDATE или DELETE. Все права могут быть даны опцией ALL.

Для вызова процедуры в приложении пользователь должен иметь права на EXECUTE.

Пользователи могут получить разрешение выдавать права другим пользователям передачей списка права по списку <userlist>, который задается опцией WITH GRANT OPTION. Пользователь может выдавать другим только те права, которыми располагает сам.

Права могут быть даны всем пользователям опцией PUBLIC на месте списка имен пользователей. Указание опции PUBLIC распространяется только на пользователей, но не на объекты базы данных.

Таблица А.52. *Права, предоставляемые пользователям и объектам базы данных*

Право	*Позволяет пользователям*
ALL	SELECT, DELETE, INSERT, UPDATE и EXECUTE
SELECT	выбирать строки из таблицы или обзора
DELETE	удалять строки из таблицы или обзора
INSERT	вставлять строки в таблицу или обзор
UPDATE	изменять строки в таблице или обзоре. Может быть задано для определенного набора столбцов
EXECUTE	выполнять хранимую процедуру

Права могут быть ликвидированы пользователем, выдавшим их, используя команду REVOKE. Если права были выданы с помощью ALL, то и ликвидированы они могут быть только в режиме ALL, если права были выданы с помощью PUBLIC, то и ликвидированы они могут быть только в режиме PUBLIC.

СИНТАКСИС

```
GRANT {
 {ALL [PRIVILEGES] / SELECT / DELETE / INSERT
        / UPDATE [ (LIST_col) ] }
              ON [TABLE] {tablename / viewname}
              TO {<object> / <userlist>}
        / EXECUTE ON PROCEDURE procname
              TO {<object> / <userlist>}
        };

<object>::= PROCEDURE procname / TRIGGER trigname / VIEW
viewname
/ [USER] username / PUBLIC /, <object>7

<userlist>::= [USER] username {, [USER] username ...7

[WITH GRANT OPTION/
```

Таблица А.53. *Синтаксические конструкции команды GRANT*

Аргумент	***Описание***
col	Имя столбца, на который выдаются права
tablename	Имя существующей таблицы, на которую распространяются права
viewname	Имя существующего обзора, на который распространяются права
<object>	Имя существующего объекта базы, на который распространяются права
<userlist>	Список пользователей, которым передаются права
WITH GRANT OPTION	Разрешает дальнейшую передачу прав пользователям, перечисленным в списке **<userlist>**

ПРИМЕР

Следующая команда передает права пользователю на SELECT и DELETE. Опция WITH GRANT OPTION дает право на дальнейшую их передачу.

```
GRANT SELECT, DELETE ON TBOOK TO MISHA WITH GRANT OPTION;
```

Данная команда дает право на выполнение процедуры PBUTHOR процедуре PBOOKAUTHOR и пользователю.

```
GRANT EXECUTE ON PROCEDURE PBUTHOR
      TO PROCEDURE PBOOKAUTHOR, MISHA;
```

INSERT

ОПИСАНИЕ

INSERT добавляет одну или несколько новых строк данных в существующую таблицу или обзор. Значения вставляются в строку в порядке столбцов в таблице, если не указан явно необязательный список вставляемых столбцов. Если список вставляемых столбцов - подмножество доступных столбцов, то заданные по умолчанию или NULL-значения автоматически устанавливаются во всех неперечисленных столбцах. Если необязательный список целевых столбцов опущен, предложение VALUES должно задавать значения для всех столбцов в таблице.

Для вставки одиночной строки данных предложение VALUES должно включать список вставляемых значений.

Для вставки множества строк, необходимо задать выражение выборки <select_expr>, которое выбирает существующие данные из других таблиц, чтобы вставить в данную. Выбранные столбцы должны соответствовать столбцам, перечисленным в списке для вставки.

Принципиально допустимо выбирать данные из той же таблицы, в которую осуществляется вставка, но такая практика не рекомендуется, потому что это может приводить к бесконечным вставкам строки.

Для выполнения команды необходимо наличие соответствующих прав. Выдача и аннулирование прав осуществляется командами GRANT и REVOKE.

СИНТАКСИС

```
INSERT [TRANSACTION transaction] INTO <object>
[(LIST_col)]
{VALUES (LIST_<val>) | <select_expr>};

<object>::= tablename / viewname

<val>::= {
<constant> / <expr>/ <function> / NULL / USER
      } [COLLATE collation]
```

Опция COLLATE недопустима для BLOB данных.

```
<constant>;:= num | "string" | charsetname "string"
```

<expr> - любое допустимое SQL-выражение, которое в качестве результата возвращает единственное значение.

```
<function>::= {
CAST (<val> AS <datatype>)
       / UPPER (<val>)
       / GEN_ID (generator, <val>)
       }
```

<select_expr> - SELECT, возвращающая несколько (возможно 0) строк, причем количество столбцов в SELECT соответствует количеству вставляемых столбцов.

Таблица А'.54. *Синтаксические конструкции команды INSERT*

Аргумент	*Описание*
INTO <object>	Имя существующей таблицы или обзора, куда вставляются данные
col	Имя существующего столбца в таблице или обзоре, в который вставляются значения
VALUES (<val> [, <val> ...])	Список значений, вставляемых в таблицу или обзор. Значения должны перечисляться в том же порядке, что и столбцы, в которые они вставляются
<select_expr>	Запрос, возвращающий строки значений для вставки в указанные столбцы таблицы или обзора
TRANSACTION transaction	Задает имя транзакции, в рамках которой выполняется команда

ПРИМЕР

Следующая команда добавляет строку к таблице, присваивая значения двум столбцам:

```
INSERT INTO TREADER (RDNUMB, RDNAME)
 VALUES ('1111-98', 'Пугачева А.Б.');
```

Следующая команда задает значения для **вставки в таблицу, используя** результаты выборки командой SELECT:

```
INSERT INTO TREADER (RDNUMB, RDNAME)
 SELECT CAST(AUTHOR as VARCHAR(8)) || '-00', AUNAME FROM
TAUTHOR
  where AUNAME<'Д';
```

INSERT CURSOR

ОПИСАНИЕ

INSERT CURSOR записывает BLOB-данные в столбец. Размер данных, записываемых из модулей приложения, не должен превышать размер сегмента BLOB. Перед вставкой данных в BLOB-курсор необходимо:

- объявить локальную переменную для хранения вставляемых данных;
- объявить длину переменной bufferlen;
- объявить BLOB-курсор для вставки (INSERT) и открыть его.

Каждая команда вставки (INSERT) в столбец BLOB вставляет текущее содержимое буфера. Между вставками необходимо заполнять буфер новыми данными. Команду INSERT следует повторять, пока не будут вставлены все данные.

INSERT CURSOR требует наличия прав на вставку (INSERT), управляемую командами GRANT и REVOKE.

Эта инструкция может использоваться в SQL.

СИНТАКСИС

```
INSERT CURSOR cursor
VALUES (:buffer [INDICATOR] :bufferlen);
```

Таблица А.55. *Синтаксические конструкции команды INSERT CURSOR*

Аргумент	*Описание*
cursor	Имя BLOB-курсора
VALUES	Конструкция, содержащая имя и длину переменной-буфера для вставки
:buffer	Имя переменной базового языка - переменной-буфера, содержащей вставляемую информацию
INDICATOR	Индикатор, указывающий, что за ним размещается длина данных, помещаемых в буфер
:bufferlen	Длина в байтах буфера для вставки

ПРИМЕР

Следующая команда внедренного SQL демонстрирует вставку в BLOB-курсор BlobCr данных из переменной wBuf, размер данных задается переменной wLen:

```
EXEC SQL
  INSERT CURSOR BlobCr VALUES (:wBuf INDICATOR :wLen);
```

OPEN

ОПИСАНИЕ

OPEN выполняет поиск в соответствии с условиями в команде DECLARE CURSOR. Выбранные строки становятся активным набором для курсора.

Курсор - это односторонний указатель в упорядоченном наборе строк, полученном с помощью конструкции SELECT команды DECLARE

CURSOR. Курсор допускает только последовательный доступу к выбранным строкам. Имеются четыре связанных команды курсора:

Таблица А.56. *Перечень команд для работы с курсором*

№ п/п	*Команда*	*Назначение*
1	DECLARE CURSOR	Объявляет курсор. Конструкция SELECT определяет состав столбцов курсора
2	OPEN	Получает строки, указанные в DECLARE CURSOR. Полученные строки образуют активный набор курсора
3	FETCH	Получает очередную строку активного набора, начиная с первой. Последовательное выполнение FETCH продвигает курсор по набору
4	CLOSE	Закрывает курсор и освобождает системные ресурсы

Эта инструкция может использоваться в SQL и DSQL.

СИНТАКСИС

SQL form:

```
OPEN [TRANSACTION transaction] cursor;
```

DSQL form:

```
OPEN [TRANSACTION transaction] cursor [USING SQL
DESCRIPTOR xsqlda]
```

Blob form:

```
OPEN [TRANSACTION name7 cursor
{INTO | USING} :blob_id;
```

Подробнее см. OPEN (BLOB).

Таблица А.57. *Синтаксические конструкции команды OPEN*

Аргумент	*Описание*
TRANSACTION transaction	Имя транзакции, в рамках которой выполняется команда OPEN

Аргумент	*Описание*
Cursor	Имя ранее объявленного курсора
USING DESCRIPTOR xsqlda	Передает значения в расширенной области дескриптора (XSQLDA) в соответствии с подготовленными параметрами команды

ПРИМЕР

```
EXEC SQL
DECLARE Mcursor CURSOR FOR
SELECT ...;
. . .

EXEC SQL
OPEN Mcursor;
. . .
// Открытие цикла чтения
. . .
EXEC SQL
FETCH Mcursor INTO ...;
// Проверка на конец выборки и выход из цикла
. . .
// Окончание цикла чтения
EXEC SQL
      CLOSE Mcursor;
```

OPEN (BLOB)

ОПИСАНИЕ

OPEN подготавливает ранее объявленный курсор для чтения или вставки BLOB-данных. В зависимости от того, что объявляет команда DECLARE CURSOR: чтение (READ BLOB) или запись (INSERT BLOB), значение blob_id обрабатывается командой OPEN по-разному.

Для конструкции READ BLOB значение blobid поступает из курсора внешней таблицы. Для конструкции INSERT BLOB это значение возвращается системой.

Эта команда может использоваться в SQL.

СИНТАКСИС

```
OPEN [TRANSACTION name] cursor
{INTO / USING} :blob_id;
```

Таблица А.58. *Синтаксические конструкции команды OPEN (BLOB)*

Аргумент	*Описание*
TRANSACTION name	Специфицирует транзакцию, в рамках которой открывается курсор. Если не указан, то применяется транзакция по умолчанию
Cursor	Имя BLOB-курсора
INTO | USING	В зависимости от типа BLOB-курсора применяется одна из конструкций: INTO: For INSERT BLOB; USING: For READ BLOB
blob_id	Идентификатор BLOB-столбца

ПРИМЕР

Следующая команда внедренного SQL объявляет и открывает BLOB-курсор.

```
EXEC SQL
BEGIN DECLARE SECTION;
BASED ON TBOOK.REFERAT blob_id;
BASED ON TBOOK.REFERAT.SEGMENT blob_segment_buf;
BASED ON TBOOK.UNIKEY key;
unsigned short blob_seg_len;
EXEC SQL
END DECLARE SECTION;

. . .

EXEC SQL
DECLARE BLOBCURSOR CURSOR FOR
INSERT INTO TBOOK;

. . .

EXEC SQL
OPEN BLOBCURSOR INTO :blob_id;
```

PREPARE

ОПИСАНИЕ

PREPARE подготавливает команду DSQL для многократного использования. Проверяет ее на отсутствие синтаксических ошибок. Опре-

деляет типы данных для произвольно указанных динамических параметров. Оптимизирует выполнение команды.

Компилирует команду для выполнения с помощью EXECUTE.

PREPARE является частью группы команд, которые подготавливают команды DSQL к выполнению.

Таблица А.59. *Перечень команд подготовки и выполнения инструкций SQL*

Команда	***Назначение***
PREPARE	Подготавливает команды DSQL к выполнению
DESCRIBE	Заполняет XSQLDA данными о команде
EXECUTE	Выполняет подготовленную ранее команду
EXECUTE IMMEDIATE	Подготавливает команду DSQL, выполняет и освобождает ее

После подготовки команда доступна для многократного выполнения в течение текущего сеанса. Чтобы подготовить и выполнить команду только один раз, можно использовать EXECUTE IMMEDIATE.

Команда устанавливает символическое имя для фактической подготовки команды DSQL. Оно не объявляется как переменная базового языка. Кроме программ С, GPRE не делает различий между верхним и нижним регистрами в командах, обрабатывая "B" и "b" как один и тот же символ. В программах на С для активации чувствительности к регистру во время предварительной обработки используется переключатель gpre-either_case.

Если используется необязательный режим INTO, команда PREPARE заполняет также в списке выбранных столбцов подготовленной команды в области расширенного SQL дескриптора (XSQLDA) данные об их типах, длине и именах.

Вместо конструкции INTO для заполнения списка выбора в XSQLDA можно использовать команду DESCRIBE.

Конструкция FROM специфицирует для подготовки с помощью PREPARE реальной команды DSQL. Она может быть переменной базового языка или символьной строкой в кавычках. Подготавливаемая команда DSQL может быть любой командой объявления данных, манипулирования данными или управления транзакциями.

Эта команда может использоваться в SQL.

СИНТАКСИС

```
PREPARE [TRANSACTION transaction] statement
```

*[***INTO** SQL DESCRIPTOR **xsqlda***]* FROM *{*:<variable> / "string"*}*;

Таблица А.60. *Синтаксические конструкции команды PREPARE*

Аргумент	*Описание*
transaction	Имя транзакции, в рамках которой выполняется команда
statement	Устанавливает псевдоним (алиас) для подготовленного предложения, которое может быть использовано в последовательности команд DESCRIBE и EXCUTE
INTO xsqlda	Задает XSQLDA, которая будет заполняться описаниями выбираемых столбцов в подготовленной команде
:<variable> "string"	Текст подготавливаемой командой PREPARE команды DSQL. Может быть переменной базового языка или символьной строкой

ПРИМЕР

Следующая команда внедренного SQL готовит команду DSQL из переменной базового языка buf. Поскольку она использует необязательное предложение INTO, можно предположить, что команда DSQL в переменой базового языка является командой SELECT.

```
EXEC SQL
       PREPARE Q INTO xsqlda FROM :buf;
```

Данная команда может быть подготовлена и описана следующим образом:

```
EXEC SQL
       PREPARE Q FROM :buf;
EXEC SQL
       DESCRIBE Q INTO SQL DESCRIPTOR xsqlda;
```

REVOKE

ОПИСАНИЕ

REVOKE ликвидирует права доступа к объектам базы данных. Права - это действия с объектом, которые разрешены пользователю. SQL-права описаны в следующей таблице.

Таблица А.61. *Перечень прав, ликвидируемых по команде REVOKE*

Право	*Запрещает пользователям*
ALL	использовать SELECT, DELETE, INSERT, UPDATE и EXECUTE
SELECT	выбирать строки из таблицы или обзора
DELETE	удалять строки из таблицы или обзора
INSERT	вставлять строки в таблицу или обзор
UPDATE	изменять строки в таблице или обзоре. Может быть задано для определенного набора столбцов
EXECUTE	выполнять хранимую процедуру
GRANT OPTION FOR	делегировать права другим пользователям

Следует отметить ограничения при использовании команды REVOKE. Ликвидировать права может только тот пользователь, кто их выдал. Одному пользователю могут быть переданы одни и те же права на объект базы данных от любого числа разных пользователей. Команда REVOKE влечет за собой лишение выданных ранее именно этим пользователем прав. Права, выданные всем пользователям опцией PUBLIC, могут быть ликвидированы командой REVOKE только с опцией PUBLIC.

СИНТАКСИС

```
REVOKE [GRANT OPTION FOR] {

{ALL [PRIVILEGES] | SELECT / DELETE / INSERT
                  / UPDATE [(col [, col ...])]}
                 ON [TABLE] {tablename / viewname}
                 FROM {<object> / <userlist>}
       / EXECUTE ON PROCEDURE procname
                 FROM {<object> / <userlist>7
       ;,•

<object> = PROCEDURE procname / TRIGGER trigname / VIEW
viewname
       / [USER] username / PUBLIC [, <object>]

<userlist> = [USER]username [, [USER] username ...]
```

Таблица А.62. *Синтаксические конструкции команды REVOKE*

Аргумент	***Описание***
GRANT OPTION FOR	Отменяет у пользователей, перечисленных в <userlist>, права на делегирование прав на объекты базы данных. Неприменим по отношению к объектам (<object>) базы данных
col	Имя столбца, на который выдаются права
tablename	Имя существующей таблицы, на которую распространяются права
viewname	Имя существующего обзора, на который распространяются права
<object>	Имя существующего объекта базы, на который распространяются права
<userlist>	Список пользователей, которым передаются права

ПРИМЕР

Следующая команда ликвидирует у пользователя права на чтение таблицы:

```
REVOKE SELECT ON TBOOK FROM MISHA;
```

Следующая команда отменяет право на выполнение процедуры PBUTHOR у процедуры PBOOKAUTHOR и пользователя:

```
REVOKE EXECUTE ON PROCEDURE PBUTHOR
       FROM PROCEDURE PBOOKAUTHOR, MISHA;
```

ROLLBACK

ОПИСАНИЕ

ROLLBACK отменяет все команды по изменению данных (DML), произведенные после последней команды COMMIT. Команды DDL (описания данных) записываются автоматически, поэтому ROLLBACK к ним неприменима.

СИНТАКСИС

```
ROLLBACK [WORK];
```

Таблица А.63. *Синтаксические конструкции команды REVOKE*

Аргумент	*Описание*
WORK	Необязательное слово; используется для совместимости

ПРИМЕР

Следующие команды иллюстрируют применение ROLLBACK в ISQL. Первая команда ROLLBACK отменяет вставку данных (команду INSERT), но не создание таблицы (CREATE TABLE), вторая - не оказывает никакого действия, поскольку результаты команды INSERT уже были сохранены (выдана команда COMMIT).

```
CREATE TABLE T (A INT);
INSERT INTO T VALUES (5);
ROLLBACK;
INSERT INTO T VALUES (7);
COMMIT;
ROLLBACK;
SELECT * FROM T;
```

SELECT

ОПИСАНИЕ

SELECT получает данные из таблиц, обзоров и хранимых процедур. Различные варианты команды SELECT позволяют:

- Получить отдельную строку или ее часть из таблицы. Такой вариант команды SELECT называют однострочным (singleton select).
- Получить множество строк или их частей из таблицы (многострочный SELECT).
- Получить логически связанные строки или части строк из соединения двух или нескольких таблиц.
- Получить логически связанные строки или части строк из объединения двух или большего числа таблиц.

Все команды SELECT содержат две обязательные конструкции (SELECT, FROM) и могут содержать комбинацию из нескольких необязательных конструкций (WHERE, GROUP BY, HAVING, UNION, PLAN, ORDER BY).

Конструкции SELECT и FROM требуются как в однострочной, так и в многострочной команде SELECT. Остальные конструкции являются необязательными. Назначение отдельных конструкций SELECT приведено в таблице.

Таблица А.64. *Основные синтаксические конструкции команды SELECT*

Конструкция	*Назначение*
SELECT	Задает список выбираемых столбцов
FROM	Идентифицирует таблицы, обзоры или хранимые процедуры, из которых выбираются данные
WHERE	Специфицирует условия выборки данных. Конструкция WHERE может содержать собственные команды SELECT, называемые подзапросами
GROUP BY	Группирует строки на основе одинаковых значений в столбцах. Может использоваться вместе с конструкцией HAVING
HAVING	Используется для задания дополнительных условий выборки из списка строк, полученного применением конструкции GROUP BY
UNION	Объединяет результаты двух или нескольких команд SELECT, образуя единую динамическую таблицу без повторяющихся строк
ORDER BY	Задает порядок сортировки возвращаемых командой SELECT строк: по возрастанию (ASC) или убыванию (DESC) значений в указанных столбцах
PLAN	Задает план выполнения запроса, который будет использован оптимизатором запросов, вместо того, который был бы выбран автоматически

СИНТАКСИС

```
SELECT [DISTINCT / ALL] {* / <val> [, <val> ...]}
FROM <tableref> [, <tableref> ... 7
       [WHERE <search_condition>]
       /"GROUP BY col /"COLLATE collation./ [, col [COLLATE
collation] . . . ]
       [HAVING <search_condition>]
       [UNION <select_expr>]
       [PLAN <plan_expr>]
       [ORDER BY <order_list>]

<val> = {
col [<array_dim>] / <constant> / <expr> / <function>
```

```
        / NULL / USER / RDB$DB_KEY
}

<array_dim>;;= [LIST_<dim>]
```

Квадратные скобки при описание массива являются ***частью синтаксиса, а не признаком необязательности.***

```
<dim>::= x[:y]
<constant>: .= пшп / "string" j charsetname "string"

<expr> - любое допустимое выражение в SQL, возвращающее
единственное значение (кроме массива и BLOB).

<function> = {
COUNT (* / [ALL] <val> / DISTINCT <val>)
        / SUM ( [ALL] <val> / DISTINCT <val>)
        / AVG ( [ALL] <val> / DISTINCT <val>)
        / MAX ( [ALL] <val> / DISTINCT <val>)
        / MIN ( [ALL] <val> / DISTINCT <val>)
        / CAST (<val> AS <datatype>)

/ UPPER (<val>)
        / GEN_ID (generator, <val>)
        }

<tableref>::= <joined_table> / table / view / procedure
[(<val> [, <val> ...])] [alias]

<joined_table>::= <tableref> <join_type> JOIN <tableref>
ON <search_condition> / (<joined_table>)

<join-type>::= {[INNER] / {LEFT / RIGHT / FULL }
[OUTER] } JOIN

<search_condition> = {<val> <operator>

{<val> / (<select_one>) }
        / <val> [NOT] BETWEEN <val> AND <val>
        / <val> [тот] LIKE <val> [ESCAPE <val>7
```

```
      | <val> [NOT 7 IN (<val> [, <val> ...] \ <se-
lect_list>)
      / <val> IS [NOT] MULL
      / <val> {[NOT7 {= | < | >} | >= |<=}
            {ALL / SOME / ANY} (<select_list>)
      / EXISTS (<select_expr>)
      / SINGULAR (<select_expr>)

/ <val> [NOT] CONTAINING <val>
      / <val> [NOT] STARTING [WITH] <val>
      / (<search_condition>)
      / NOT <search_condition>
      / <search_condition> OR <search_condition>
      / <search_condition> AND <search_condition>7

<operator>: := {= j < /> /<= />= / !< / !> /<> / '•=}

<select_one> - SELECT, возвращающая один столбец в одной
строке.

<select_list> - SELECT, возвращающая один столбец в не-
скольких (возможно 0) строках.

<select_expr> - SELECT, возвращающая несколько столбцов в
нескольких (возможно 0) строках.

<plan_expr>::=
[JOIN / [SORT] MERGE] (<plan_item> / <plan_expr>
            [, <plan_item> / <plan_expr> ...])

<plan_item>: := {table / alias}
      NATURAL / INDEX (<index> [, <index> ...]) / ORDER
<index>

<order_list>::=
{col / int} [COLLATE collation] [ASC [ENDING] /
DESC [ENDING J]
[, <order_list>]
```

Таблица А.65. *Синтаксические конструкции команды SELECT и их опции*

Аргумент	*Описание*
SELECT [DISTINCT | ALL]	Задает перечень выбираемых данных. Опция DISTINCT запрещает выборку повторяющихся значений. Опция ALL (по умолчанию) выбирает любые значения
{* | <val> [, <val> ...]	* задает режим выборки всех столбцов указанных таблиц. <val> [, <val> ...] задает выборку указанных столбцов и выражений
FROM <tableref> [, <tableref>...]	Задает список таблиц, обзоров и хранимых процедур, из которых осуществляется выборка данных. Список может содержать присоединения, и эти присоединения могут быть вложенными
table	Имя существующей в базе таблицы
view	Имя существующего в базе обзора
procedure	Имя существующей в базе хранимой процедуры, функционирующей аналогично команде SELECT
alias	Алиас - альтернативное имя для таблицы или обзора. После объявления в <tableref> алиас может использоваться для указания ссылок на таблицу или обзор
<joined_table>	Указание на таблицу, содержащую конструкцию JOIN
<join_type>	Тип используемого соединения. По умолчанию принимается INNER
WHERE <search_cond>	Задает условие выборки строк
GROUP BY <col> [, <col>	Разбивает множество выбранных строк на группы, содержащие все строки с одинаковыми значениями в перечисленных столбцах
COLLATE collation	Задает порядок сравнения данных, получаемых по запросу
HAVING <search_cond>	Используется вместе с конструкцией GROUP BY. Задает дополнительные условия отбора результирующих строк

Аргумент	*Описание*
UNION	Задает объединение результатов двух или нескольких выборок с идентичным составом столбцов. Каждая из выборок может осуществляться из различных таблиц, обзоров или хранимых процедур
PLAN <plan_expr>	Задает план выполнения запроса, который будет использован оптимизатором запросов, вместо того, который был бы выбран автоматически
<plan_item>	Указывает табличные и индексные методы для плана
ORDER BY <order_list>	Задает порядок сортировки строк (по значениям столбцов, перечисленных в <order_list>)

ПРИМЕР

Следующая команда выбирает, используя шаблон *, все столбцы из таблицы:

```
SELECT * FROM TREADER;
```

Следующая команда выбирает перечисленные столбцы из таблицы и упорядочивает выборку:

```
SELECT RDNUMB ,RDNAME FROM TREADER ORDER BY RDNAME;
```

Следующая команда выбирает перечисленные столбцы из двух таблиц (перечень читателей и взятых ими книг), используя для связи третью таблицу, и упорядочивает выборку по фамилиям:

```
SELECT TREADER.rdnumb, TREADER.rdname, TBOOK.booknm
 FROM TREADER, TBOOK, TBOOK_READER
 where tbook_reader.reader=treader.unikey and
            tbook_reader.bookkey=tbook.unikey
 order by rdname;
```

Следующая команда использует в качестве дополнительного условия подзапрос (список книг невостребованных читателями):

```
SELECT BOOKNM,unikey from TBOOK
 where matherkey!=0 and
  not EXISTS
```

```
(select * from TBOOK_READER where bookkey=TBOOK.unikey)
```

Следующая команда использует контекстный поиск:

```
SELECT BOOKNM,unikey from TBOOK
 where BOOKNM CONTAINING 'C++'
```

Следующая команда использует агрегатную функцию SUM для подсчета количества строк в таблице по каждой из групп строк, задаваемых полем RDNAME. Полученная выборка дополнительно фильтруется, оставляя в результате только те строки, которым в исходных таблицах соответствует более чем одна строка. В данном случае выбирается перечень читателей, взявших более 1 книги. Для уточнения из каких именно таблиц выбираются данные, используются алиасы.

```
select a.RDNAME, SUM(1) sumbook
 FROM TREADER a, TBOOK_READER b
 where (a.UNIKEY=b.READER) GROUP BY RDNAME
HAVING SUM(1)>1;
```

Следующая команда задает левое соединение, используя подзапрос:

```
SELECT a.rdnumb, a.rdname, b.booknm
 FROM TREADER a LEFT JOIN TBOOK b on
 EXISTS(SELECT * FROM TBOOK_READER ab
 where ab.reader=a.unikey and ab.bookkey=b.unikey);
```

Следующая команда явно задает план для выполнения запроса, рассмотренного в третьем по счету примере:

```
SELECT TREADER.rdnumb, TREADER.rdname, TBOOK.booknm
 FROM TREADER, TBOOK, TBOOK_READER
 where tbook_reader.reader=treader.unikey and
                tbook_reader.bookkey=tbook.unikey
PLAN sort(JOIN(TBOOK_READER NATURAL,TBOOK INDEX
(RDB$PRIMARY4),TREADER INDEX (RDB$PRIMARY9)))
  order by rdname;
```

Следующая команда использует пользовательскую функцию для преобразования текста заданного столбца, содержащего кириллицу в верхний регистр:

```
SELECT  rupper(BOOKNM) from TBOOK
 where unikey=6 or unikey=7;
```

Функция объявлена, как:

```
DECLARE EXTERNAL FUNCTION RUPPER
    CSTRING (256)
    RETURNS CSTRING (256)
    ENTRY_POINT '_srupper' MODULE_NAME 'MYDLL.dll';
```

SET DATABASE

ОПИСАНИЕ

SET DATABASE объявляет дескриптор базы для указанной базы данных и связывает его с этой базой, что позволяет сделать необязательной спецификацию различных баз данных и во время проектирования и во время выполнения. Приложения, которые обращаются к нескольким базам данных одновременно, должны использовать команды SET DATABASE, чтобы установить отдельные дескрипторы для каждой из баз.

dbhandle - определенное приложением имя дескриптора базы данных. Обычно имя дескриптора - это сокращение фактического имени базы данных. Однажды объявленный, дескриптор может использоваться в последовательностях команд CONNECT, COMMIT, ROLLBACK. Дескрипторы могут также использоваться в транзакциях, чтобы разделить таблицы, имеющие одинаковые имена в разных базах данных.

"<dbname>" является платформно-зависимой файловой спецификацией базы данных, связанной с dbhandle. Она должна следовать соглашениями синтаксиса файла для сервера, где находится база данных.

GLOBAL, STATIC и EXTERN - необязательные параметры, которые определяют область действия объявления базы данных. Область действия имени по умолчанию - GLOBAL - означает, что дескриптор базы данных доступен всем модулям кода в приложении. STATIC означает, что дескриптор базы данных доступен в том модуле кода, где он объявлен. EXTERN ссылается на глобальный дескриптор базы данных, объявленный в другом модуле.

Необязательные параметры COMPILETIME и RUNTIME дают возможность одному дескриптору базы данных относиться к одной базе данных, когда приложение подготавливается, и к другой базе данных, когда приложение выполняется пользователем. Если опущены или если указана только база данных COMPILETIME, то InterBase будет использовать указанную базу и во время подготовки и во время выполнения.

Параметры USER (имя пользователя) и PASSWORD (пароль) требуются для всех приложений клиента PC, но не обязательны для других удаленных соединений. Имя пользователя и пароль проверяются сервером в базе данных защиты перед разрешением соединения удаленным приложениям.

Эта команда может использоваться в SQL.

СИНТАКСИС

```
SET {DATABASE / SCHEMA} dbhandle =
[GLOBAL / STATIC / EXTERN]
        [COMPILETIME] [FILENAME] "<dbname>"
        [USER "<name>" PASSWORD "<string>"]
        [RUNTIME [FILENAME] {"<dbname>" / :var}
        [USER {"<name>" / :var} PASSWORD {"<string>" /
:var}]];
```

Таблица А.66. *Синтаксические конструкции команды SET DATABASE*

Аргумент	*Описание*
dbhandle	Задает алиас (псевдоним) для указанной базы данных. Алиас должен быть уникальным внутри программы. Он используется в командах SQL, поддерживающих дескрипторы баз данных
GLOBAL	Область действия имени по умолчанию - GLOBAL означает, что дескриптор базы данных доступен всем модулям кода в приложении
STATIC	STATIC означает, что дескриптор базы данных доступен в том модуле кода, где он объявлен
EXTERN	Ссылается на глобальный дескриптор базы данных, объявленный в другом модуле
COMPILETIME	Указывает базу данных, используемую для просмотра столбцов на подготовительном этапе. Если в конструкции SET DATABASE указана только одна база, она будет использоваться и на этапе выполнения
"<dbname>"	Задает имя и путь к базе данных, заданных дескриптором. Задание параметра зависит от платформы, на которой реализуется база
RUNTIME	Указывает на другую базу данных для использования на этапе выполнения, отличную от используемой на подготовительном этапе
:<var>	Переменная базового языка, содержащая спецификацию базы, имя пользователя, пароль

Аргумент	*Описание*
USER "<name>"	Требуется для клиентского PC-соединения, не обязательна для других. Обеспечивает контроль имени пользователя на сервере, где размещается база данных. Используется вместе с паролем (PASSWORD) для контроля прав доступа пользователя на сервере
PASSWORD "<string>"	Требуется для клиентского PC-соединения, не обязательна для других. Обеспечивает контроль пароля на сервере, где размещается база данных. Используется вместе с именем пользователя (USER "<name>") для контроля прав доступа пользователя на сервере

ПРИМЕР

Следующая команда внедренного SQL объявляет дескриптор базы данных:

```
EXEC SQL
       SET DATABASE DB1 = "testbase.gdb";
```

Для корректной работы препроцессора, ему необходимо подсоединиться к базе данных. В то же время, сами данные ему не нужны. Таким образом, препроцессор может использовать одну базу, а работающее приложение другую.

Следующая команда внедренного SQL объявляет две базы данных: для подготовительного этапе и этапе выполнения. Для указания базы данных этапа выполнения используется переменная базового языка WorkBase:

```
EXEC SQL
 SET DATABASE EMDBP = "testbase.gdb" RUNTIME : WorkBase;
```

SET GENERATOR

ОПИСАНИЕ

SET GENERATOR инициализирует начальное значение для вновь созданного генератора или сбрасывает значение существующего. Генератор обеспечивает формирование уникального последовательного числового значение функцией GEN_ID (). Если вновь созданный генератор не инициализирован с помощью SET GENERATOR, его значения по умолчанию - 0. Int задает новое значение для генератора. Когда GENID () используется для вставки или модификации значений в столбцах, то ее

возвращаемое значение равно int плюс приращение - параметр шага, указанный в GEN_ID ().

Чтобы указать, что первое выдаваемое значение было равно 1, следует указать в SET GENERATOR стартовое значение равным 0 и установить значение шага в функции GEN_ID() в 1.

Следует проявлять осторожность при переустановке генератора, который поддерживает создание уникальных значений для первичных или уникальных ключей - такая переустановка может привести к появлению дублирующихся значений.

СИНТАКСИС

```
SET GENERATOR name TO int;
```

Таблица А.67. *Синтаксические конструкции команды SET GENERATOR*

Аргумент	*Описание*
Name	Имя существующего генератора
Int	Значение, которое задается генератору в качестве стартового, должно лежать в диапазоне от -2^{31} до $2^{31}-1$

ПРИМЕР

```
SET GENERATOR SYSNUMBER TO 1000;
```

Если функция GEN_ID() вызовет генератор **с шагом 1, то первый** возвращенный номер будет 1001.

SET NAMES

ОПИСАНИЕ

SET NAMES задает набор символов для использования при последующих соединениях с базой данных. Это дает возможность ISQL переопределить заданный по умолчанию набор символов для базы данных.

Команда SET NAMES должна появляться перед командой CONNECT, на которую она должна воздействовать.

СИНТАКСИС

```
SET NAMES [charset];
```

Таблица А.68. *Синтаксические конструкции команды SET NAMES*

Аргумент	*Описание*
Charset	Имя набора символов, идентифицирующего активный набор символов для данного процесса. Значение по умолчанию - NONE

ПРИМЕР

```
SET NAMES WIN1251;
CONNECT "testbase.gdb";
```

SET STATISTICS

ОПИСАНИЕ

SET STATISTICS дает возможность вычислить селективность индекса. Индексная селективность - число, основанное на количестве различных строк в таблице, которое используется оптимизатором InterBase при обращении к таблице. Индексная селективность используется оптимизатором для определения плана выполнения запроса, исходя из минимизации ожидаемого количества просматриваемых строк таблицы. В таблицах, где число двойных значений в индексированных столбцах радикально увеличивается или уменьшается, периодическое вычисление индексной селективности может существенно ускорить время выполнения.

Только создатель индекса может использовать команду SET STATISTICS.

Команда SET STATISTICS не перестраивает индекс. Для перестройки индекса следует использовать команду ALTER INDEX.

СИНТАКСИС

```
SET STATISTICS INDEX name;
```

Таблица А.69. *Синтаксические конструкции команды SET STATISTICS INDEX*

Аргумент	*Описание*
name	**Имя** существующего индекса, для которого рассчитывается **индексная** селективность

ПРИМЕР

```
SET STATISTICS INDEX TREADER_RDNAME;
```

SET TERM

ОПИСАНИЕ

Определяет символ или символы для индикации конца команд. Значение по умолчанию - точка с запятой (;).

СИНТАКСИС

```
SET TERM string;
```

Таблица АЛО. *Синтаксические конструкции команды SET TERM*

Аргумент	*Описание*
string	Определяет символ или символы для индикации конца команд. Значение по умолчанию - точка с запятой (;)

По умолчанию ISQL команды должны заканчиваться точкой с запятой (;). Чтобы изменить символ индикации конца используется команда SET TERM.

SET TERM обычно используется с командой CREATE PROCEDURE или CREATE TRIGGER. Процедуры и триггеры определяются, используя язык процедур и триггеров, в котором команды всегда заканчиваются точкой с запятой. Процедура или триггер в целом сами должны быть завершены символом, отличным от точки с запятой.

Текстовый файл, содержащий определение CREATE PROCEDURE или CREATE TRIGGER, должен включать одну команду SET TERM перед определениями и одну команду SET TERM после определений. Сначала SET TERM определяет новый символ завершения; в конце - восстанавливает точку с запятой (;), как значение по умолчанию.

Использование SET TERM из isql сеанса дает тот же эффект, что и использование признака конца в командной строки.

ПРИМЕР

Следующий пример содержит использование команды SET TERM при создании хранимой процедуры. Первая команда SET TERM задает ## как символ окончания; завершающая команда SET TERM восстанавливает ";" в качестве символа окончания.

```
SET TERM ^ ;
CREATE PROCEDURE PAUTHOR (
    P1 CHAR (1),
    P2 CHAR (1))
RETURNS (
```

```
    AUTHOR INTEGER,
    AUNAME VARCHAR (60))
AS
begin
 for select author, auname from tauthor
  into :author, :auname
 do
  if(auname>p1) then
  if(auname<p2) then suspend;
end ^
SET TERM ; ^
```

SET TRANSACTION

ОПИСАНИЕ

SET TRANSACTION определяет заданный по умолчанию доступ к базе данных из транзакции, обработку конфликта блокировки и уровень взаимодействия с другими параллельно работающими транзакциями, обращающимися к тем же самым данным. Команда может также резервировать блокировки для таблиц.

По умолчанию, транзакция имеет к базе доступ на чтение и запись. Если транзакция должна только читать данные, следует задать параметр read only.

Когда одновременные транзакции пытаются модифицировать одни и те же данные в таблицах, только первая получает к ним доступ. Никакие другие транзакции не могут модифицировать или удалять эти данные, пока первая не завершена или не отменена. По умолчанию, транзакции ждут окончания выполняющейся транзакции, затем делают попытку их собственных операций. Чтобы заставить транзакцию немедленно завершиться и сообщить об ошибке конфликта блокировки без ожидания, указывается параметр NO WAIT.

ISOLATION LEVEL определяет, как заданная по умолчанию транзакция взаимодействует с другими транзакциями, обращающимися к тем же таблицам. Заданный по умолчанию уровень изоляции - SNAPSHOT (снимок). Это обеспечивает повторный просмотр базы данных в момент старта транзакции. Изменения, сделанные другими транзакциями, невидимы. SNAPSHOT TABLE STABILITY обеспечивает повторное чтение базы данных, гарантируя, что транзакции не могут записывать в таблицы, хотя они могут читать из них.

READ COMMITTED дает возможность заданной по умолчанию транзакции видеть последние изменения, сделанные другими транзакциями. Можно также модифицировать строки, пока не происходит никаких конфликтов модификаций. Нейтральные изменения, сделанные другими транзакциями, остаются невидимыми, пока не будет выдана команда COMMIT. Режим READ COMMITTED имеет также два необязательных параметра:

1. NO RECORD_VERSION (значение по умолчанию) читает только самую последнюю версию строки. Если определена опция, разрешающая блокировку ожидания (WAIT), то транзакция ждет, пока самая последняя версия строки не будет сохранена или восстановлена (rolled back), и повторяет ее чтение.

2. RECORD_VERSION читает самую последнюю завершенную версию строки, даже если более новая (но незавершенная) версия также постоянно находится на диске.

Предложение RESERVING дает возможность транзакции регистрировать желательный уровень доступа для указанных таблиц при ее запуске, а не когда делается попытка выполнения операций на таблицах. Резервирование таблиц при старте транзакций может уменьшить вероятность возникновения тупиков.

СИНТАКСИС

```
SET TRANSACTION [READ WRITE / READ ONLY]
[WAIT / NO WAIT]
      / [ISOLATION LEVEL] {SNAPSHOT [TABLE STABILITY]
            / READ COMMITTED [[NO] RECORD_VERSION] }]
      [RESERVING <reserving_clause>;

<reserving_clause>::= table [, table ...]
[FOR [SHARED / PROTECTED] {READ / WRITE}]
[,<reserving_clause>]
```

Таблица А.71. *Синтаксические конструкции команды SET TRANSACTION*

Аргумент	*Описание*
READ WRITE	Специфицирует, что транзакция может читать и писать таблицы (по умолчанию)
READ ONLY	Специфицирует, что транзакция может только читать таблицы
WAIT	Специфицирует, что транзакция ждет доступа, если возникает конфликт блокировок с другой транзакцией (по умолчанию)
NO WAIT	Специфицирует, что транзакция немедленно завершается, если возникает конфликт блокировок с другой транзакцией

Аргумент	*Описание*
ISOLATION LEVEL	Специфицирует уровень изоляции для данной транзакции при попытке доступа к таблицам одновременно с другими транзакциями (по умолчанию SNAPSHOT)
RESERVING <reserving_clause>	Резервирует перечень возможных блокировок таблиц при старте транзакций

ПРИМЕР

Следующая команда устанавливает заданную по умолчанию транзакцию (gds_$trans) с уровнем изоляции READ COMMITTED. Если транзакция сталкивается с конфликтом модификаций, она ждет и перехватит управление сразу после завершения или отката блокирующей транзакции.

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

Следующая команда резервирует три таблицы:

```
SET TRANSACTION
      ISOLATION LEVEL READ COMMITTED
      NO RECORD_VERSION WAIT
      RESERVING TABLE1, TABLE2 FOR SHARED WRITE,
            TABLE3 FOR PROTECTED WRITE;
```

UPDATE

ОПИСАНИЕ

Команда UPDATE изменяет одну или несколько существующих строк в таблице или обзоре. Необязательное предложение WHERE используется, чтобы ограничить модификации требуемым подмножеством строк в таблице. UPDATE не может модифицировать часть массива. Без опции WHERE команда модифицирует все строки таблицы.

При обновлении столбца BLOB команда UPDATE заменяет содержимое BLOB новым значением.

Права на выполнение UPDATE регулируются командами GRANT и REVOKE.

СИНТАКСИС

```
UPDATE {table / view}
SET col = <val> [, col = <val> ...]
  [WHERE <search_condition>];
```

```
<val>::= {
col [<array_dim>] | <constant> | <expr> / <function>
      / NULL / USER


<array_dim>::= [LIST_<dim>]
```

Здесь квадратные скобки являются *частью синтаксиса описания массивов, а не элементом метаязыка.*

```
<dim>::= x[:y]
<constant>::= num / "string" | charsetname "string"

<expr> - любое допустимое в SQL выражение, возвращающее
единственное значение (кроме массивов и BLOB).

<function>::= {
CAST (<val> AS <datatype>)
      / UPPER (<val>)
      / GEN_ID (generator, <val>)
      ;

<search_condition>::= {<val> <operator> {<val> / (<select_one>)}
      | <val> [NOT] BETWEEN <val> AND <val>
      / <val> [NOT] LIKE <val> /"ESCAPE <val>]
      / <val> [NOT] IN (<val> [, <val> ...] / <se-
lect_list>)

/ <val> IS [NOT] NULL
      / <val> {[NOT] {= | < | >} | >= | <=}
            {ALL / SOME / ANY} (<select_list>)
      / EXISTS (<select_expr>)
      / SINGULAR (<select_expr>)
      / <val> [NOT] CONTAINING <val>
      / <val> [NOT] STARTING [WITH] <val>
      / (<search_condition>)
      / NOT <search_condition>
      / <search_condition> OR <search_condition>
      / <search_condition> AND <search_condition>}
```

Таблица **А.72.** *Синтаксические конструкции команды UPDATE*

Аргумент	*Описание*
table | view	Имя существующей таблицы или обзора
SET col = <val>	Специфицирует изменяемые столбцы и присваиваемые им значения
WHERE <search_cond>	Определяет условия, при которых производится модификация строк

ПРИМЕР

Следующая инструкция изменяет столбец для всех строк в таблице:

```
UPDATE TBOOK SET
 NUM_PRESENCE = NUM_ALL;
```

Следующая инструкция использует предложение WHERE, чтобы ограничить модификацию столбца подмножеством строк:

```
UPDATE TREADER SET RDNUMB='1278-98',
 RDNAME=' Гребенкина Н.'
where UNIKEY=39;
```

WHENEVER

ОПИСАНИЕ

WHENEVER перехватывает SQLCODE ошибки и предупреждения. Каждая выполнимая команда SQL возвращает значение SQLCODE, чтобы указать ее успех или неудачу. Если SQLCODE равен 0, команда завершилась успешно. Ненулевое значение указывает на ошибку, предупреждение или регистрирует состояние «не найдено». Если соответствующее состояние возникло, WHENEVER позволяет, используя метку GOTO, перейти к подпрограмме обработки ошибок в приложении или, используя CONTINUE, игнорировать ошибку.

WHENEVER может ограничить размер приложения, потому что приложение может использовать один набор подпрограмм для обработки всех ошибок и предупреждений.

WHENEVER-команды должны предшествовать любой команде SQL, которая может приводить к ошибке. Каждое перехватываемое состояние требует отдельной команды WHENEVER. Если WHENEVER опущена для какого-либо состояния, то оно не перехватывается.

Команды WHENEVER . . . CONTINUE должны быть в начале программы обработки ошибок, чтобы предупредить возникновение бесконечных циклов в программах.

Эта команда может использоваться в SQL.

СИНТАКСИС

```
WHENEVER {NOT FOUND / SQLERROR / SQLWARNING}
{GOTO label / CONTINUE};
```

Таблица А.73. *Синтаксические конструкции команды WHENEVER*

Аргумент	*Описание*
NOT FOUND	Перехватывает SQLCODE = 100: не найдено указанных строк в выполненной команде
SQLERROR	Перехватывает SQLCODE < 0: аварийное завершения команды
SQLWARNING	Перехватывает SQLCODE > 0 AND < 100: системное предупреждение или информационное сообщение
GOTO label	Переход к указанной точке программы при обнаружении ошибки или предупреждения
CONTINUE	Игнорирует ошибки и предупреждения и пытается продолжить программу

ПРИМЕР

В следующем фрагменте программы внедренного SQL три команды WHENEVER определяют, к какой метке следует перейти при возникновении предупреждения или ошибки:

Переход к метке при возникновении любой ошибки.

```
EXEC SQL
  WHENEVER SQLERROR GO TO Error;
```

Переход к метке при возникновении ситуаций типа «конец данных» (SQLCODE=100).

```
EXEC SQL
  WHENEVER NOT FOUND GO TO AllDone;
```

Игнорируются любые ошибки.

```
EXEC SQL
  WHENEVER SQLWARNING CONTINUE;
```

А.2. Функции

Таблица А.74. *Перечень стандартных функций InterBase*

Функция	*Тип*	*Назначение*
AVG()	Агрегатная	Вычисляет среднее значение
CAST()	Преобразование	Преобразует данные одного типа в другой
COUNT()	Агрегатная	Подсчитывает количество строк, удовлетворяющих условию
GEN_ID()	Числовая	Возвращает сгенерированное системой значение
MAX()	Агрегатная	Находит минимальное значение
MIN()	Агрегатная	Находит минимальное значение
SUM()	Агрегатная	Вычисляет сумму
UPPER()	Преобразование	Преобразует строку в верхний регистр

Агрегатные функции производят вычисления на множестве значений, таких как множество значений столбца, полученного командой SELECT.

Функции преобразования либо преобразуют данные из одного типа в другой, либо переводят символьные данные в верхний регистр.

Числовая функция GEN_ID() выдает сгенерированный системой номер, который может быть вставлен в столбец, требующий числовой тип данных.

AVG()

ОПИСАНИЕ

AVG() - агрегатная функция, которая возвращает среднее **число** из множества значений в указанном столбце или выражении. AVG() применима только к числовым типам данных.

Если значение одного из полей, по которым вычисляется среднее, NULL, то оно автоматически исключается из вычислений. Автоматическое исключение предотвращает искажение средних от включения бессмысленных данных. AVG() вычисляется по диапазону выбранных строк. Если число строк, возвращенных командой SELECT, равно О, AVG() возвращает значение NULL.

СИНТАКСИС

AVG (*[*ALL*]* <val> / DISTINCT <val>) ;

\ Таблица А.75. *Синтаксические конструкции функции AVG()*

Аргумент	*Описание*
ALL	Возвращает среднее по всем величинам
DISTINCT	Исключает повторяющиеся величины перед расчетом
<val>	Столбец или выражение, по которому вычисляется среднее

ПРИМЕР

Следующая команда иллюстрирует использование функции AVG() для выбранного множества строк таблицы с приведением исходных данных с помощью функции CAST к типу double precision для корректного подсчета:

```
SELECT AVG(CAST(BNUMBER as double precision))
 FROM  TBOOK_PLACE
 WHERE PLACEKEY=43;
```

CAST()

ОПИСАНИЕ

CAST() позволяет смешивать числовые и символьные данные в одном выражении, преобразуя <val> к указанному типу данных. Обычно только однотипные данные могут сравниваться в условиях поиска. CAST() может использоваться в условиях поиска, преобразуя один тип данных в другой для сравнения. Допустимые преобразования показаны в таблице.

Таблица А.76. *Типы данных, преобразуемых функцией CAST()*

Исходный тип данных	*Результирующий тип данных*
NUMERIC	CHARACTER, VARYING CHARACTER, DATE
CHARACTER, VARYING CHARACTER	NUMERIC, DATE
DATE	CHARACTER, VARYING CHARACTER, DATE

Если преобразование невозможно, генерируется сообщение об ошибке.

СИНТАКСИС

```
CAST (<val> AS <datatype>);
```

Таблица А.77. *Синтаксические конструкции функции CAST()*

Аргумент	*Описание*
<val>	Столбец, константа или выражение, которое преобразуется к символьному типу
<datatype>	Тип данных, к которому производится преобразование

ПРИМЕР

Следующая команда иллюстрирует использование функции CAST для приведения исходных данных к типу double precision, обеспечивая корректный подсчет средних значений с помощью функции AVG() по выбранному множеству строк таблицы:

```
SELECT AVG(CAST(BNUMBER as double precision))
 FROM  TBOOK_PLACE
 WHERE PLACEKEY=43;
```

GEN_ID()

ОПИСАНИЕ

Функция GEN_ID():

- увеличивает текущее значение указанного генератора на заданную величину (шаг),
- возвращает текущее значение указанного генератора.
- GEN_ID() удобно использовать для автоматической генерации уникальных значений столбцов первичного (PRIMARY KEY) или уникального ключа (UNIQUE). Для вставки сгенерированного значения в столбец можно использовать триггер, процедуру или команду SQL, вызывающую функцию GEN_ID().

Генератор создается командой CREATE GENERATOR. По умолчанию, начальное значение генератора 0. Другое значение можно установить командой SET GENERATOR.

СИНТАКСИС

```
GEN_ID (generator, step);
```

Таблица А.78. *Синтаксические конструкции функции GEN_ID ()*

Аргумент	*Описание*
generator	Имя существующего генератора
step	Целое или выражение, задающее приращение для увеличения или уменьшения значения генератора. Значение лежит в диапазоне от -2^{31} до $2^{31}-1$

ПРИМЕР

Следующее определение триггера включает запрос к GEN_ID ():

```
SET TERM !! ;
CREATE TRIGGER I_TREADER_1 FOR TREADER ACTIVE
BEFORE INSERT POSITION 0
as
begin
  if (new.UNIKEY is NULL) then
new.UNIKEY=GEN_ID(sysnumber,1);
  if (new.RDNUMB is NULL) then exception NO_RDNUMB;
  if (new.RDNAME is NULL) then exception NO_RDNAME;
end
SET TERM ; !!
```

При первом включении триггера NEW.RDNUMB получит значение 1, затем 2 и т.д.

MAX()

ОПИСАНИЕ

MAX() - агрегатная функция, которая возвращает наибольшее число из множества значений в указанном столбце или выражении. Если значение одного из полей, по которым вычисляется среднее, NULL, то оно автоматически исключается из вычислений. MAX() вычисляется по диапазону выбранных строк. Если число строк, возвращенных командой SELECT, равно О, MAX() возвращает значение NULL.

Если MAX() применяется для столбцов типа CHAR, VARCHAR или текстовый BLOB, максимум определяется в зависимости от символьного набора (CHARACTER SET) и порядка сравнения (COLLATION).

СИНТАКСИС

MAX ([ALL] <val> / DISTINCT <val>);

Таблица А.79. *Синтаксические конструкции функции MAX()*

Аргумент	*Описание*
ALL	Поиск ведется среди всех значений в столбце
DISTINCT	Повторяющиеся значения исключаются перед поиском наибольшего
<val>	Столбец или выражение, которое преобразуется к числовому типу

ПРИМЕР

Следующая команда иллюстрирует использование функций SUM(), MIN() и MAX() для множества строк таблицы:

```
SELECT SUM(BNUMBER), MAX(BNUMBER), MIN(BNUMBER)
FROM TBOOK_PLACE;
```

MIN()

ОПИСАНИЕ

MIN() - агрегатная функция, которая возвращает наименьшее число из множества значений в указанном столбце или выражении. Если значение одного из полей, по которым вычисляется среднее, NULL, то оно автоматически исключается из вычислений. MIN() вычисляется по диапазону выбранных строк. Если число строк, возвращенных командой SELECT, равно О, MIN() возвращает значение NULL.

Если MIN() применяется для столбцов типа CHAR, VARCHAR или текстовый BLOB, минимум определяется в зависимости от символьного набора (CHARACTER SET) и порядка сравнения (COLLATION).

СИНТАКСИС

MIN (*[***ALL***]* <val> / **DISTINCT** <val>) ;

Таблица А.80. *Синтаксические конструкции функции MAX()*

Аргумент	*Описание*
ALL	Поиск ведется среди всех значений в столбце
DISTINCT	Повторяющиеся значения исключаются перед поиском наименьшего
<val>	Столбец или выражение, которое преобразуется к числовому типу

ПРИМЕР

Следующая команда иллюстрирует использование функций SUM(), MIN() и MAX() для множества строк таблицы:

```
SELECT SUM(BNUMBER), MAX(BNUMBER), MIN(BNUMBER)
FROM TBOOK_PLACE;
```

SUM()

ОПИСАНИЕ

SUM() - агрегатная функция, которая возвращает сумму из множества значений в указанном столбце или выражении. Если значение одного из полей, по которым вычисляется среднее, NULL, то оно автоматически исключается из вычислений. SUM() вычисляется по диапазону выбранных строк. Если число строк, возвращенных командой SELECT, равно О, SUM() возвращает значение NULL.

СИНТАКСИС

SUM (*[**ALL**]* <val> / **DISTINCT** <val>) ;

Таблица А.81. *Синтаксические конструкции функции SUMQ*

Аргумент	*Описание*
ALL	Сумма значений в столбце
DISTINCT	Повторяющиеся значения исключаются перед расчетом суммы
<val>	Столбец или выражение, которое преобразуется к числовому типу

ПРИМЕР

Следующая команда иллюстрирует использование функций SUM(), MIN() и MAX() для множества строк таблицы:

```
SELECT SUM(BNUMBER), MAX(BNUMBER), MIN(BNUMBER)
FROM TBOOK_PLACE;
```

UPPER()

ОПИСАНИЕ

UPPER() преобразует заданную строку к верхнему регистру.

Если в указанном символьном наборе нет чувствительности к регистру, то функция UPPER() не изменяет данные.

СИНТАКСИС

UPPER (<val>);

Замечание. Функция UPPER не работает с кириллицей. Для соответствующих преобразований следует использовать функцию пользователя (UDF), выполняющую аналогичные действия.

Таблица А.82. *Синтаксические конструкции функции UPPER()*

Аргумент	*Описание*
<val>	Столбец, константа или выражение, преобразуемое к символьному типу

ПРИМЕР

Если поле ABC содержит строку «asdfghjk» , то результатом вызова функции UPPER будет «ASDFGHJK».

```
UPPER (BMatthews);
```

Если поле ABC содержит строку «фывапрол», то в результатом следующего вызова функции UPPER будет «фывапрол». Для преобразования кириллицы следует использовать пользовательскую функцию типа RUPPER. См. пример 6.27.

А.3. Типы данных

InterBase поддерживает большинство SQL типов данных, хотя версии до 6 прямо и не поддерживает SQL DATE, TIME, TIMESTAMP. Дополнительно к стандартным типам данных InterBase поддерживает BLOB (binary large object) и массивы любых, кроме BLOB, типов.

Ниже перечислены типы данных, допустимых в командах SQL для InterBase.

Таблица А.83. *Типы данных, используемые в InterBase*

Наименование	*Размер*	*Диапазон/ точность*	*Описание*
BLOB	переменный		BLOB хранит большие объемы данных, такие как графика, тексты, цифровой звук и т.д. Основной структурный элемент - сегмент. Подтип описывает содержание
CHAR(n); CHARACTER (n)	n символов	1-32767 байт	Размер символа в наборе определяет максимальное число символов (общая длина не больше 32K)
DATE	64 бит	от 1.01.100	Дополнительно включает и данные о времени для версии младше 6.
TIME	64 бит	от 0 до 23.59.9999	Время (только в версии от 6).
TIMESTAMP	64 бит	от 1.01.100	Дополнительно включает и данные о времени (только в версии от 6).
DECIMAL (precision, scale)	переменный	1-15/1-15 точность не превосходит длину	Число с указанным количеством десятичных цифр. Например: DECIMAL(10, 3) ppppppp.sss.
DOUBLE PRECISION	64 бит	От $1.7*10^{-308}$ до $1.7*10^{308}$	15 значащих цифр. Реальный размер зависит от платформы - обычно 64 бит
FLOAT	32 бит	От $3.4*10^{-38}$ до $3.4*10^{38}$	7 значащих цифр

Наименование	*Размер*	*Диапазон/ точность*	*Описание*
INTEGER	32бит	От -2^{31} до $2^{31}-1$	Длинное целое
NUMERIC (precision, scale)	переменный	1-15/1-15 точность не превосходит длину	Число с указанным количеством десятичных цифр. Например: NUMERIC(10, 3) ppppppp.sss.
SMALLINT	16бит	От -2^{15} до $2^{15}-1$	Короткое целое
VARCHAR(n); VARYING CHAR (n); VARYING CHARACTER (n)	n символов	1-32767байт	Строки переменной длины. Размер символа в наборе определяет максимальное число символов (общая длина не больше 32K)

A.4. Коды завершения SQL

По завершении любой выполнимой команды SQL формируется ее код завершения (SQLCODE), сигнализирующей об успешности ее выполнения.

В таблице приведены формируемые коды завершения (SQLCODE).

Таблица А.84. *Коды завершения SQL в InterBase*

SQLCODE	*Сообщение*	*Описание*
< 0	SQLERROR	Произошла ошибка. Команда не выполнена
0	SUCCESS	Успешное выполнение
+1-99	SQLWARNING	Системное предупреждение или информационное сообщение
+100	NOT FOUND	Нет строк для обработки или достигнут конец текущего активного набора

В ISQL при возникновении ошибки выдается ее номер и соответствующее сообщение.

ПРИЛОЖЕНИЕ Б

Сообщения об ошибках

Каждое сообщение об ошибке сопровождается соответствующим кодом SQLCODE, перечень кодов приведен в таблице.

Таблица Б.1. *Сообщения об ошибках SQLCODE (сводные)*

SQLCODE	*Описание*
0	Успешное завершение
1-99	Предупреждение или информационное сообщение
100	Конец файла (списка)
< 0	Ошибка. Команда **не** выполнена

С одним кодом SQLCODE связано одно или несколько сообщений, сам текст сообщения однозначно идентифицируется внутренним кодом InterBase. В текст сообщения могут также включаться дополнительные данные, указывающие на объекты базы, к которым относится само сообщение:

<string> - строка, например имя объекта базы или типа объекта;

<long> - длинное целое, например идентификационный номер объекта базы или типа объекта;

<digit> - целое, например идентификационный номер объекта базы или типа объекта.

Например:

- ошибка с кодом «-84» сопровождается сообщением «Procedure <string> does not return any values» (код InterBase - 335544668)
- ошибка с кодом «-104» сопровождается сообщением «Token unknown—line <long>, char <long> (код InterBase - 335544634)
- ошибка с кодом «-692» сопровождается сообщением «Maximum indexes per table (<digit>) exceeded» (код InterBase - 335544477).

Вставляемые тексты или числовые значения позволяют конкретизировать объект, при работе с которым возникла ошибка.

Полный перечень кодов ошибок (SQLCODE), кодов ошибок InterBase и связанных с ними сообщений приведен в системной документации (Language Reference).

Краткое описание кодов ошибок (SQLCODE) и комментарии к ним приведены в таблице Б.2.

В первом столбце приведен код ошибки (SQLCODE), в последнем - перечень связанных с ним внутренних кодов InterBase.

Как коды SQLCODE, так и однозначные внутренние коды InterBase можно использовать в функциях обработки ошибок InterBase, чтобы выдавать сообщения или выполнять иные действия, основанные на этих кодах. Использование внутренних кодов позволяет точнее идентифицировать ситуацию, однако при этом нужно помнить о непереносимости таких программ на иные платформы.

Внутренние коды доступны через массив состояния в приложениях с внедренным SQL.

SQLCODE коды доступны и в приложениях с внедренным SQL, и в процедурах и триггерах (см. конструкцию WHEN … DO).

Таблица Б.2. *Сообщения об ошибках SQLCODE*

SQLCODE	***Комментарий***	***Коды InterBase***
101	Попытка прочесть в буфер больше данных, чем их есть, может использоваться для индикации конца данных, например при чтении BLOB.	335544366L
100	Попытка прочесть данные после их исчерпания (конец данных)	335544338, 335544354, 335544367, 335544374
-84	Нарушение прав доступа	335544554, 335544555, 335544668
-103	Неверный тип данных для константы	335544571
-104	Синтаксическая ошибка в команде SQL	335544343, 335544390, 335544425, 335544426, 335544429, 335544440, 335544456, 335544570, 335544579, 335544590, 335544591, 335544592, 335544634, 335544608, 335544612

SQLCODE	Комментарий	Коды InterBase
-150	Ошибка обновления данных	335544360, 335544362, 335544446, 335544546
-151	Попытка обновить столбец с доступом только на чтение	335544359
-155	< Строка > не является именем базовой таблицы для обзора	335544658
-157	Необходимо указать имя столбца в выражении select для обзора	335544598
-158	Количество столбцов не соответствует списку в команде select	335544599
-162	Ключ не доступен для обзора, основанного на нескольких таблицах	335544685
-170	Ошибка в параметрах процедуры или пользовательской функции	335544512,335544619
-171	Синтаксическая ошибка, несогласованные данные	335544439, 335544458, 335544618
-172	Функция < Строка > не определена	335544438
-204	Данные, специфицированные в команде, не определены в базе (сообщение может уточняться именем данных) или неверно указаны	335544463, 335544502, 335544509, 335544511, 335544515, 335544516, 335544532, 335544551, 335544568, 335544573, 335544580, 335544581, 335544588, 335544589, 335544589, 335544595, 335544620, 335544621, 335544622, 335544635, 335544636, 335544640, 335544662,
-205	Не определен указанный в команде столбец	335544396, 335544552
-206	Неверен или не определен столбец	335544578, 335544587, 335544596
-208	Ошибка в конструкции ORDER BY	335544617
-219	Таблица <строка-имя> не определена или неверна	335544395

SQLCODE	*Комментарий*	*Коды InterBase*
-239	Слишком маленький кэш	335544691
-260	Кэш переопределен	335544690
-281	Таблица <строка-имя> не указана в плане	335544637
-282	Таблица <строка-имя> встречается неоднократно, необходимо задавать алиасы	335544638, 335544643, 335544659, 335544660
-283	Таблица <строка-имя> встречается в плане, но нигде не упомянута в запросе	335544639
-284	Индекс <строка-имя> не может использоваться в плане	335544642
-291	Столбец, используемый в первичном или уникальном ключе (PRIMARY или UNIQUE), должен быть объявлен, как NOT NULL	335544531
-292	Нельзя обновить ограничение (RDB$REF_CONSTRAINTS)	335544534
-293	Нельзя обновить ограничение (RDB$CHECK_CONSTRAINTS)	335544535
-294	Нельзя удалить CHECK ограничение (RDB$CHECK_CONSTRAINTS)	335544536
-295	Нельзя обновить ограничение (RDB$RELATION_CONSTRAINTS)	335544545
-296	Внутренняя проверка - ошибка в RDB$CONSTRAINT_TYPE	335544547
-297	Команда нарушает ограничение целостности (CHECK) для таблицы или обзора < строка-имя >	335544558
-313	Несоответствие между количествами элементов в списках столбцов и переменных	335544669
-314	Не удается выполнить транслитерацию между символьными наборами (кодовыми таблицами)	335544565
-401	Неверный оператор сравнения	335544647

SQLCODE	*Комментарий*	*Коды InterBase*
-402	Недопустимая операция при работе с BLOB	335544368, 335544414, 335544427
-406	Нижний индекс за пределами допустимых границ	335544457
-407	Сегмент уникального ключа (UNIQUE KEY) - Null	335544435
-413	Ошибка при преобразовании данных	335544334, 335544454
-501	Ошибка при работе с дескриптором	335544327, 335544577
-502	Повторное объявление или открытие курсора	335544574, 335544576
-504	Курсор не определен	335544572
-508	Нет записи для выполнения операции fetch	335544348
-510	Курсор не является обновляемым	335544575
-518	Неопределенный запрос	335544582
-519	Команда PREPARE выполняется над открытым курсором	335544688
-530	Защита FOREIGN KEY <строка> или невозможность выполнить подготовку CREATE DATABASE или SCHEMA	335544466, 335544597
-532	Ошибка ввода-вывода при выполнении транзакции	335544469
-551	Не разрешен доступ <строка> к <строка>	335544352
-552	Недостаточно прав для выполнения операции	335544550, 335544553
-553	Нельзя изменить права пользователя	335544529
-595	Текущая позиция разрушена	335544645
-596	В начале потока обнаружена неверная операция	335544644
-597	Длина предшествующего файла не определена, <строка> должна включать стартовую позицию	335544632

SQLCODE	Комментарий	Коды InterBase
-598	Номер файла тени (shadow) должен быть положительным	335544633
-599	Узел не поддерживается	335544607
-600	Ошибка при работе с узлом	335544625, 335544680
-601	База данных или файл не существуют	335544646
-604	Ошибка размерности массива	335544593, 335544594
-605	Некорректная ссылка столбца на себя	335544682
-607	Некорректная модификация метаданных	335544351, 335544549, 335544657
-615	Конфликт блокировки данных	335544475, 335544476, 335544507
-616	Запрет модификации (удаления) метаданных из-за их использования другими объектами базы	335544530, 335544539, 335544540, 335544541, 335544543, 335544630, 335544674
-617	Нельзя изменить триггер, переименовать столбец из-за их использования в других ограничениях логической целостности	335544542, 335544544
-618	Нельзя удалить (изменить) сегмент индекса из-за их использования в других ограничениях логической целостности	335544537, 335544538
-625	Ошибка при проверке правильности данных столбца <строка>, значение "<строка>"	335544347
-637	Повторная спецификация <строка> не поддерживается	335544664
-660	Ошибка (недопустимая спецификация) при создании индекса или внешнего ключа	335544533, 335544628
-663	Неверная спецификация индекса	335544624, 335544631, 335544672
-664	Размер ключа для индекса <строка> больше допустимого	335544434

SQLCODE	Комментарий	Коды InterBase
-677	Ошибка расширения <строка>	335544445
-685	Ошибка при работе со строкой или массивом	335544465, 335544670, 335544671
-689	Неверный тип для страницы (возможно с указанием номера)	335544403, 335544650
-690	Сегмент недопустим в индексном выражении <строка>	335544679
-691	Новый размер записи <число> слишком велик	335544681
-692	Превышено максимальное число индексов для таблицы	335544477
-693	Слишком много параллельных заданий на выполнение одного и того же запроса	335544663
-694	Нет доступа к столбцу <строка-имя> в обзоре <строка-имя>	335544684
-802	Исключительная ситуация при арифметической операции, переполнения или усечение строкового данного	335544321
-803	Попытка ввода дублей в уникальный ключ, защита уникального или первичного ключа <строка-имя>	335544349, 335544665
-804	Несоответствие в количестве аргументов или несоответствии числа переменных и числа столбцов	335544380, 335544583, 335544584, 335544586
-806	В конструкции VIEW WITH CHECK OPTION разрешены только простые имена столбцов	335544600
-807	Не указана конструкция WHERE для VIEW WITH CHECK OPTION	335544601
-808	Для VIEW WITH CHECK OPTION можно использовать только одну таблицу	335544602
-809	Для VIEW WITH CHECK OPTION нельзя использовать конструкции: DISTINCT, GROUP или HAVING	335544603

SQLCODE	Комментарий	Коды InterBase
-810	Для VIEW WITH CHECK OPTION нельзя использовать подзапросы	335544605
-811	Несколько строк в однострочном запросе (singleton select)	335544652
-816	Внешний файл нельзя открыть для вывода данных	335544651
-817	Попытка выполнить неподдерживаемую операцию (запись в объект только для чтения)	335544361, 335544371, 335544444
-820	Работа с устаревшими версиями	335544356, 335544379, 335544437, 335544467
-823	Неверный дескриптор закладки	335544473
-824	Неверный уровень блокировки <цифра>	335544474
-825	Неверный дескриптор блокировки	335544519
-826	Неверный дескриптор оператора	335544585
-827	Неверный направление в операции поиска	335544655
-828	Неверная позиция ключа	335544678
-829	Неверная ссылка на столбец	335544616
-830	Столбец используется в агрегате	335544615
-831	Попытка создать второй первичный ключ (PRIMARY KEY) к одной и той же таблице	335544548
-832	Количества столбцов в FOREIGN KEY и PRIMARY KEY не совпадают	335544604
-833	Вычисление подобных выражений не поддерживается	335544606
-834	Диапазон номеров для обновления <число> не найден	335544508
-835	Неверная контрольная сумма	335544649
-836	Исключение <цифра>	335544517
-837	Рестарт разделенного менеджера кэша	335544518

SQLCODE	***Комментарий***	***КодыInterBase***
-838	База данных <строка-имя> закрывается через <цифра> секунд	335544560
-839	Неверный формат файла журнала	335544686
-840	Переполнение файла журнала	335544687
-841	Слишком много версий	335544677
-842	Неверно указаны размер или точность числовых данных	335544697, 335544698, 335544699, 335544700, 335544701
-901	Ошибка при попытке выполнения команды	**335544322,** 335544326, 335544326, 335544328, 335544329, 335544330, **335544331,** 335544332, 335544337, 335544339, 335544340, 335544341, 335544342, 335544345, 335544350, 335544353, 335544355, 335544357, 335544358, 335544363, 335544364, 335544365, 335544369, 335544370, 335544372, 335544376, **335544377,** 335544378, 335544382, 335544383, 335544392, 335544407, 335544408, 335544418, 335544419, 335544420, 335544428, **335544431,** 335544442, 335544443, 335544450, **335544468,** 335544485, 335544510, 335544559, 335544561, 335544562, 335544563, 335544609, 335544610, 335544611,335544613, 335544614, 335544623, 335544626, **335544627,** 335544641, 335544656, 335544666, 335544673, 335544675

SQLCODE	***Комментарий***	***КодыInterBase***
-902	Ошибка при выполнении команды (возможно внутренняя ошибка InterBase)	**335544333,** 335544335, 335544344, **335544346,** 335544373, 335544384, 335544385, 335544387, 335544388, 335544394, 335544397, 335544398, 335544399, 335544400, 335544401, 335544402, 335544404, 335544405, 335544406, 335544409, **335544410, 335544411,** 335544412, 335544413, 335544415, 335544416, 335544417, 335544422, 335544423, **335544432,** 335544436, 335544448, 335544449, 335544470, 335544471, **335544472,** 335544478, 335544479, 335544480, 335544481, 335544483, 335544506, 335544520, 335544528, 335544557, 335544653, 335544654, 335544564, 335544569
-904	**Ошибка** при **выполнении запроса**	335544324, 335544375, 335544381, 335544386, 335544389, 335544391, **335544393,** 335544424, 335544430, 335544451, 335544453, 335544455, 335544460, 335544661, 335544676, 335544683
-906	Продукт <строка-имя> не лицензирован	335544452
-909	Команда Drop database выполнена с ошибками	335544667
-911	Запись в транзакции <номер> "in limbo"	335544459
-913	"Deadlock"	335544336

SQLCODE	***Комментарий***	***Коды InterBase***
-922	Файл **<строка-имя>** не является именем базы данных	335544323
-923	Ошибка соединения	335544421, 335544461, 335544462
-924	Ошибка при подключении	335544325, 335544441, 335544648
-926	Откат не выполнен	335544447
-999	Внутренняя ошибка InterBase	335544689

Приложение В

Описание структуры базы данных TESTBASE

База TESTBASE.GDB используется для примеров.

Домены

Таблица В.1. *Перечень доменов*

Наименование	*Тип данных*	*Ограничения*
PrmKey:	Integer	NOT NULL
D_ELEM	CHAR(2)	CHECK (value in ('H', 'Li', 'Na', 'K'))

Таблицы

Таблица В.2. *Структура* ***TBOOK****. (Описание книги)*

Имя поля	*Тип (домен)*	*Ограничения*	*Описание*
UniKey	PrmKey		Первичный автоинкрементный ключ
Matherkey	Integer		Ключ рубрики верхнего уровня
BookNm	VarChar (250)		Наименование книги
Referat	BLOB		Реферат
Num_All	Smallint	Default 0 Not Null	Количество экземпляров

Имя поля	*Тип (домен)*	*Ограничения*	*Описание*
Num_Presence	Smallint	Default 0 Not Null	Кол-во экземпляров в наличии

Таблица 8.3. *Содержание* ***TBOOK.*** *(Описание книги)*

Unikey	Matherkey	Booknm	Num_all	Num_presence
2	0	Программирование	1	1
3	0	Учебники	1	1
4	0	Математика	1	1
5	0	Беллетристика	1	1
6	2	Макрокоманды MS Word	1	1
7	2	Word 6 for Windows	1	0
8	2	Язык С++	1	0
9	2	Введение в С++ Builder	1	1
10	2	Borland - Технологии. SQL-Link InterBase, Paradox for Windows, Delphi	1	1
11	2	С и С++ Справочник	1	1
12	2	Введение в технологию АТМ	1	0
13	3	The history of England. Absolute Monarchy	1	0
14	3	Справочник по правописанию и литературной правке	1	1
15	3	Тесты. Сборник 11 класс. Варианты и ответы государственного тестирования. Пособие для подготовки к тестированию	1	0
16	4	Математические вопросы динамики вязкой несжимаемой жидкости	1	0

Unikey	Matherkey	Booknm	Num_ all	Num_ pres- ence
17	5	Кровь нерожденных	1	1
18	5	Тайна	1	0

Примечание: В ряде примеров, чтобы не загромождать результаты вывода используется версия таблицы без двух последних столбцов.

Таблица В.4. *Структура* ***TAUTHOR****. (Авторы)*

Имя *Ноля*	***Тип (домен)***	***Ограничения***	***Описание***
Author	PrmKey		Первичный автоинкремент-ный ключ
AuName	VarChar(60)	NOT NULL	Имя автора
Comment	VarChar(80)		Примечание

Таблица В.5. *Содержание* ***TAUTHOR.*** *(Авторы)*

AUTHOR	*AUNAME*	*COMMENT*
19	Дашкова Полина	
20	Хмелевская Иоанна	
21	Ладыжинская Ольга Александровна	
22	Бурова И.И.	
23	Розенталь Д.Э.	
24	без авторов	
25	Культин Н.Б.	
26	Хаселир Райнер Г.	
27	Фаненштих Клаус	
28	Подбельский Вадим Валериевич	
29	Елманова Н.З.	
30	Кошель СП.	

AUTHOR	*AUNAME*	*COMMENT*
31	Дунаев Сергей	
32	Луис Дерк	
33	Буассо Марк	
34	Деманж Мишель	
35	Мюнье Жан-Мари	

Таблица В.6. *Структура **TPLACE.** (Описание места хранения)*

Имя поля	*Тип (домен)*	*Ограничения*	*Описание*
UniKey	PrmKey		Первичный автоинкрементный ключ
CdPlace	VarChar (25)		Библиотечный код места
TxPlace	VarChar (60)		Наименование места

Таблица В.7. *Содержание **TPLACE.** (Описание места хранения)*

UNIKEY	*CDPLACE*	*TXPLACE*
41	А-113	Комната 3, стеллаж 5, полка 6
42	Б-16	Комната 3, стеллаж 2, полка 4
43	В-54	Комната 2, стеллаж 8, полка 1
44	Г-88	Комната 4, стеллаж 1, полка 2

Таблица В.8. *Структура **TREADER.** (Описание читателей)*

Имя поля	*Тип (домен)*	*Ограничения*	*Описание*
UniKey	PrmKey		Первичный автоинкрементный ключ
RdNumb	VarChar (10)		№ читательского билета
RdName	VarChar (60)		Имя читателя

Таблица В.9. *Содержание* ***TREADER.*** *(Описание читателей)*

UNIKEY	*RDNUMB*	*RDNAME*
36	1267-89	Арцибашев С.
37	1369-99	Светлова В.
38	1456-00	Стародуб Е.
39	1273-92	Гребенкина Н.
40	1400-00	Пащенко О.
83	1401-99	Грамотный Н.Е.

Таблица В.10. *Структура* ***TBOOK_AUTHOR.*** *(Описание связи Автор–Книга)*

Имя поля	*Тип (домен)*	*Ограничения*	*Описание*
UniKey	PrmKey		Первичный автоинкрементный ключ
Author	PrmKey		Ключ для таблицы Автор
BookKey	PrmKey		Ключ для таблицы Книга
B1	COMPUTED BY	select a.auname from tauthor a where a.author=tbook_author.author	Вычисляемое поле. Добавляет имя автора
B2	COMPUTED BY	select a.booknm from tbook a where a.unikey=tbook_author.bookkey	Вычисляемое поле. Добавляет название книги

Таблица В.11. *Содержание* ***TBOOK_AUTHOR.****. (Описание связи Автор - Книга)*

UNIKEY	*AUTHOR*	*BOOKKEY*
58	25	6
59	33	12
60	34	12
61	35	12
62	32	11
63	31	10
64	29	9
65	30	9
66	28	8
67	26	7
68	27	7
69	21	16
70	24	15
71	23	14
72	22	13
73	19	17
74	20	18

Примечание: В таблице не приведены значения вычисляемых полей. В ряде примеров, чтобы не загромождать результаты вывода используется версия таблицы без вычисляемых последних столбцов.

Таблица В.12. *Структура* ***TBOOKJPLACE.*** *(Описание связи Книга-Место)*

Имя поля	*Тип(домен)*	*Ограничения*	*Описание*
UniKey	PrmKey		Первичный автоинкрементный ключ
PlaceKey	PrmKey		Ключ для таблицы Мест

Имя поля	*Тип (домен)*	*Ограничения*	*Описание*
BookKey	PrmKey		Ключ для таблицы Книга
Bnumber	SmallInt		Количество экземпляров

Таблица В.13. *Содержание Описания связи Книга - Место*

UNIKEY	*PLACEKEY*	*BOOKKEY*	*BNUMBER*
45	41	17	3
46	41	18	3
47	42	16	1
48	43	6	2
49	43	7	1
50	43	8	2
51	43	9	1
52	43	11	2
53	43	10	1
54	43	12	2
55	44	13	2
56	44	14	2
57	44	15	2

Таблица В.14. *Структура* ***TBOOK_READER****(Описание связи Книга - Читатель)*

Имя поля	*Тип (домен)*	*Ограничения*	*Описание*
UniKey	PrmKey		Первичный автоинкрементный ключ
Reader	PrmKey		Ключ для таблицы Читателей
BookKey	PrmKey		Ключ для таблицы Книга
FirstDate	Date		Дата выдачи книги

Имя поля	Тип (домен)	Ограничения	Описание
NextDate	Date		Дата продления срока пользования

Таблица В.15. *Содержание* ***TBOOK_READER*** *(Описание связи Книга - Читатель)*

UNIKEY	READER	BOOKKEY	FIRSTDATE	NEXTDATE
75	36	18	05.05.2000	06.07.2000
76	36	16	19.07.2000	19.07.2000
77	36	15	22.07.2000	22.07.2000
78	37	18	06.07.2000	06.07.2000
79	38	8	15.07.2000	15.07.2000
80	39	13	12.06.2000	12.07.2000
81	40	12	05.07.2000	05.07.2000
82	40	7	28.07.2000	28.07.2000

SQL скрипт создания базы данных

Приведенный ниже SQL скрипт создания базы данных включает помимо описания таблиц, также и описание процедур и триггеров.

В описании внешних функций (DECLARE EXTERNAL FUNCTION) описаны функции из библиотеки MyDll.dll. Текст реализации некоторых из этих функций на C++ приведен в примерах. Если Вы не хотите строить свою библиотеку DLL, исключите описание этих функций из скрипта.

```
CREATE DATABASE 'D:\Database\Testbase.gdb'
USER 'SYSDBA' PASSWORD 'masterkey'
PAGE_SIZE 1024;

CREATE DOMAIN D_ELEM AS CHAR(2) CHARACTER SET WIN1251
CHECK (value in ('H', 'Li', 'Na', 'K'));

CREATE DOMAIN PRMKEY AS INTEGER
NOT NULL
CONSTRAINT CHECK(value>=0);
```

```
CREATE GENERATOR SYSNUMBER;
SET GENERATOR SYSNUMBER TO 0;

/* Exceptions definitions */

CREATE EXCEPTION ERR_RUBRIC 'Неверно указана рубрика
для книги';

CREATE EXCEPTION NO_AUTHOR 'Не указано имя автора';

CREATE EXCEPTION NO_AUTHORKEY 'Не указан или неверно
указан автор';

CREATE EXCEPTION NO_BOOKKEY 'Не указана или неверно
указана книга';

CREATE EXCEPTION NO_BOOKNM 'Не указано наименование
книги';

CREATE EXCEPTION NO_PLACE 'Не указан шифр места хра-
нения';

CREATE EXCEPTION NO_PLACEKEY 'Не указана или неверно
указано место хранения';

CREATE EXCEPTION NO_PLACENM 'Не указано наименование
места хранения';

CREATE EXCEPTION NO_PRESENCE 'Книги нет в наличии';

CREATE EXCEPTION NO_RDNAME 'Не указано имя читателя';

CREATE EXCEPTION NO_RDNUMB 'Не указан № читательского
билета';

CREATE EXCEPTION NO_READERKEY 'Не указан или неверно
указан читатель';

CREATE EXCEPTION NO_RUBRIC 'Не указана рубрика для
книги';

DECLARE EXTERNAL FUNCTION ASCII_CHAR
    INTEGER
    RETURNS CHAR(1)
```

```
    ENTRY_POINT 'IB_UDF_ascii_char' MODULE_NAME
'ib_udf';

DECLARE EXTERNAL FUNCTION BLOB_STR
    BLOB
    RETURNS CSTRING(255) FREE_IT
    ENTRY_POINT '_blob_str' MODULE_NAME 'MYDLL';


DECLARE EXTERNAL FUNCTION B_AND
    SMALLINT,
    SMALLINT
    RETURNS SMALLINT
    ENTRY_POINT '_Sand' MODULE_NAME 'MyDLL.dll';


DECLARE EXTERNAL FUNCTION B_NOT
    SMALLINT
    RETURNS SMALLINT
    ENTRY_POINT '_Snot' MODULE_NAME 'MyDLL.dll';


DECLARE EXTERNAL FUNCTION B_OR
    SMALLINT,
    SMALLINT
    RETURNS SMALLINT
    ENTRY_POINT '_Sor' MODULE_NAME 'MyDLL.dll';


DECLARE EXTERNAL FUNCTION CLRBIT
    SMALLINT,
    SMALLINT
    RETURNS SMALLINT
    ENTRY_POINT '_Sclr' MODULE_NAME 'MyDLL.dll';


DECLARE EXTERNAL FUNCTION CNVL
    CSTRING(80),
    CSTRING(80)
    RETURNS CSTRING(80)
    ENTRY_POINT '_f_nvlc' MODULE_NAME 'MyDLL.dll';

DECLARE EXTERNAL FUNCTION DNVL
    DOUBLE PRECISION,
```

```
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION
    ENTRY_POINT '_f_nvld' MODULE_NAME 'MyDLL.dll';

DECLARE EXTERNAL FUNCTION FABS
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION
    ENTRY_POINT '_d_abs' MODULE_NAME 'MyDLL.dll';

DECLARE EXTERNAL FUNCTION FDAY
    DATE
    RETURNS INTEGER BY VALUE
    ENTRY_POINT '_f_day' MODULE_NAME 'MyDLL.dll';

DECLARE EXTERNAL FUNCTION FMONTH
    DATE
    RETURNS INTEGER BY VALUE
    ENTRY_POINT '_f_month' MODULE_NAME 'MyDLL.dll';

DECLARE EXTERNAL FUNCTION FROUND
    DOUBLE PRECISION,
    INTEGER
    RETURNS DOUBLE PRECISION
    ENTRY_POINT '_pround' MODULE_NAME 'MyDLL.dll';

DECLARE EXTERNAL FUNCTION FYEAR
    DATE
    RETURNS INTEGER BY VALUE
    ENTRY_POINT '_f_year' MODULE_NAME 'MyDLL.dll';

DECLARE EXTERNAL FUNCTION IFC
    DOUBLE PRECISION,
    VARCHAR(256),
    VARCHAR(256)
    RETURNS VARCHAR(256)
    ENTRY_POINT '_if_c' MODULE_NAME 'MyDLL.dll';

DECLARE EXTERNAL FUNCTION IFCD
    CSTRING(80),
    CSTRING(80),
    INTEGER,
    DOUBLE PRECISION,
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION
```

```
    ENTRY_POINT '_ifcd' MODULE_NAME 'MyDLL.dll';

DECLARE EXTERNAL FUNCTION IFD
    DOUBLE PRECISION,
    DOUBLE PRECISION,
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION
    ENTRY_POINT '_if_c' MODULE_NAME 'MyDLL.dll';

DECLARE EXTERNAL FUNCTION IFI
    DOUBLE PRECISION,
    INTEGER,
    INTEGER
    RETURNS INTEGER
    ENTRY_POINT '_if_c' MODULE_NAME 'MyDLL.dll';

DECLARE EXTERNAL FUNCTION INVL
    INTEGER,
    INTEGER
    RETURNS INTEGER
    ENTRY_POINT '_f_nvli' MODULE_NAME 'MyDLL.dll';

DECLARE EXTERNAL FUNCTION RTRIM
    CSTRING(80)
    RETURNS CSTRING(80) FREE_IT
    ENTRY_POINT 'IB_UDF_rtrim' MODULE_NAME 'ib_udf';

DECLARE EXTERNAL FUNCTION RUPPER
    CSTRING(256)
    RETURNS CSTRING(256)
    ENTRY_POINT '_srupper' MODULE_NAME 'MYDLL.dll';

DECLARE EXTERNAL FUNCTION SQRT
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT 'IB_UDF_sqrt' MODULE_NAME 'ib_udf';

DECLARE EXTERNAL FUNCTION SUBSTR
    CSTRING(80),
    SMALLINT,
    SMALLINT
    RETURNS CSTRING(80) FREE_IT
    ENTRY_POINT 'IB_UDF_substr' MODULE_NAME 'ib_udf';
```

```
CREATE TABLE TAUTHOR (
    AUTHOR PRMKEY,
    AUNAME VARCHAR(60) character set WIN1251 collate
PXW_CYRL,
    COMMENT VARCHAR(80) character set WIN1251 collate
WIN1251,
    AUNAMECR VARCHAR{60) character set WIN1251 col-
late WIN1251);

CREATE TABLE TBOOK {
    UNIKEY PRMKEY,
    MATHERKEY INTEGER,
    BOOKNM VARCHAR(250) character set WIN1251 collate
WIN1251,
    REFERAT BLOB sub_type 0 segment size 80,
    NUM_ALL SMALLINT DEFAULT 0 NOT NULL,
    NUM_PRESENCE SMALLINT DEFAULT 0 NOT NULL);

CREATE TABLE TBOOK_AUTHOR (
    UNIKEY PRMKEY,
    AUTHOR PRMKEY,
    BOOKKEY PRMKEY,
    B1 COMPUTED BY {(select a.auname from tauthor a
where a.author=tbook_author.author)),
    B2 COMPUTED BY ((select a.booknm from tbook a
where a.unikey=tbook_author.bookkey)));

CREATE TABLE TBOOK_PLACE (
    UNIKEY PRMKEY,
    PLACEKEY PRMKEY,
    BOOKKEY PRMKEY,
    BNUMBER SMALLINT);

CREATE TABLE TBOOK_READER (
    UNIKEY PRMKEY,
    READER PRMKEY,
    BOOKKEY PRMKEY,
    FIRSTDATE DATE,
    NEXTDATE DATE);

CREATE TABLE TPLACE (
    UNIKEY PRMKEY,
    CDPLACE VARCHAR(25) character set WIN1251 collate
WIN1251,
```

```
    TXPLACE VARCHAR(60) character set WIN1251 collate
WIN1251);

CREATE TABLE TREADER (
    UNIKEY PRMKEY,
    RDNUMB VARCHAR(25) character set WIN1251 collate
WIN1251,
    RDNAME VARCHAR(60) character set WIN1251 collate
WIN1251);

CREATE VIEW NORUBRIC1(
    UNIKEY,
    BOOKNM)
AS  select a.UNIKEY, a.BOOKNM from tbook a
   where a.matherkey>0;

CREATE VIEW NORUBRICS(
    UNIKEY,
    BOOKNM,
    B1)
AS  select a.UNIKEY, a.BOOKNM, b.B1 from tbook a,
tbook_author b  where a.matherkey>0 and
a.unikey=b.bookkey
.

CREATE VIEW RUBRICS(
    UNIKEY,
    BOOKNM)
AS  select UNIKEY, BOOKNM from tbook  where
 (matherkey=0)
;

ALTER TABLE TBOOK ADD CHECK(BOOKNM NOT IN  ('Слон',
'Моська'));
ALTER TABLE TAUTHOR ADD PRIMARY KEY  (AUTHOR);
ALTER TABLE TBOOK ADD PRIMARY KEY  (UNIKEY);
ALTER TABLE TBOOK_AUTHOR ADD PRIMARY KEY  (UNIKEY);
ALTER TABLE TBOOK_PLACE ADD PRIMARY KEY  (UNIKEY);
ALTER TABLE TBOOK_READER ADD PRIMARY KEY  (UNIKEY);
ALTER TABLE TPLACE ADD PRIMARY KEY  (UNIKEY);
ALTER TABLE TREADER ADD PRIMARY KEY  (UNIKEY);
CREATE INDEX TREADER_RDNAME ON TREADER  (RDNAME);
CREATE UNIQUE INDEX TREADER_RDNUMB ON TREADER
 (RDNUMB);
```

```
SET TERM ^ ;

CREATE PROCEDURE PAUTHOR (
    P1 CHAR(1),
    P2 CHAR(1))
RETURNS (
    AUTHOR INTEGER,
    AUNAME VARCHAR(60))
AS
begin
 for select author, auname from tauthor
  into :author, :auname
 do
  if(auname>p1) then
  if(auname<p2) then suspend;
end
л

CREATE PROCEDURE PBOOKAUTHOR
RETURNS (
    UNIKEY INTEGER,
    MATHERKEY INTEGER,
    BOOKNM VARCHAR(250),
    AUTHORS VARCHAR(250),
    REFERAT BLOB sub_type 0 segment size 80)
AS
 declare variable auname varchar(60);
 declare variable ws integer;
begin
 ws=-1;
 for select a.UNIKEY, a.matherkey, a.booknm,
    a.referat, b.auname
  from tbook a, tauthor b, tbook_author c
  where (a.unikey=c.bookkey and c.author=b.author)
  order by 1
  into :UNIKEY, :matherkey, :booknm, :referat,
:auname
  do
   begin
    if(ws!=UNIKEY) then
     begin
      if(ws!=-1) then suspend;
      ws=UNIKEY;
```

```
      authors=auname;
     end
    else
     authors=authors||', '||auname;
   end
 if(ws!=-1) then suspend;
end


CREATE PROCEDURE PBUTHOR (
    CODE INTEGER)
RETURNS (
    AUTHORS VARCHAR(250))
AS
 declare variable auname varchar(60);
 declare variable UNIKEY integer;
 declare variable ws integer;
begin
 ws=-1;
 for select a.UNIKEY, b.auname
  from tbook a, tauthor b, tbook_author c
  where (a.unikey=:Code and a.unikey=c.bookkey and
c.author=b.author)
  into :UNIKEY, :auname
  do
   begin
    if(ws=-1) then authors=auname;
    else authors=authors||', '||auname;
    ws=UNIKEY;
   end
 if(ws!=-1) then suspend;
end
л

SET TERM ; ^

SET TERM ^ ;

CREATE TRIGGER D_TBOOK_READER FOR TBOOK_READER
ACTIVE BEFORE DELETE POSITION 0
AS
BEGIN
  update tbook set num_presence=num_presence+1
  where unikey=old.bookkey;
```

```
END


CREATE TRIGGER I_TAUTHOR_1 FOR TAUTHOR
ACTIVE BEFORE INSERT POSITION 0
as
begin
  if (new.AUTHOR is NULL) then
new.AUTHOR=GEN_ID(sysnumber,1);
  if (new.AUNAME is NULL) then exception NO_AUTHOR;
  if (new.COMMENT is NULL) then new.COMMENT=" ";
end
л

CREATE TRIGGER I_TBOOK_1 FOR TBOOK
ACTIVE BEFORE INSERT POSITION 0
as
begin
  if (new.UNIKEY is NULL) then
new.UNIKEY=GEN_ID(sysnumber,1);
  if (new.MATHERKEY is NULL or new.MATHERKEY<0) then
  BEGIN
   update RDB$EXCEPTIONS SET
   Rdb$message='Не указана рубрика для <' ||
new.BOOKNM || '>'
   where Rdb$exception_name='NO_RUBRIC';
   exception NO_RUBRIC;
  END
  if (new.MATHERKEY>0) then
   if (NOT EXISTS (select * from TBOOK where (uni-
key=new.MATHERKEY)))
    then exception ERR_RUBRIC;
  if (new.BOOKNM is NULL) then exception NO_BOOKNM;
end


CREATE TRIGGER I_TBOOK_AUTHOR_1 FOR TBOOK_AUTHOR
ACTIVE BEFORE INSERT POSITION 0
as
begin
  if (new.UNIKEY is NULL) then
new.UNIKEY=GEN_ID(sysnumber,1);
  if (new.AUTHOR is NULL) then exception
NO_AUTHORKEY;
```

```
  if (new.BOOKKEY is NULL) then exception NOJBOOKKEY;
  if (NOT EXISTS(select * from TAUTHOR where
(AUTHOR=new.AUTHOR)))
   then exception NO_AUTHORKEY;
  if (NOT EXISTS(select * from TBOOK where
(UNIKEY=new.BOOKKEY)))
   then exception NO_BOOKKEY;
end
^

CREATE TRIGGER I_TBOOK_PLACE_1 FOR TBOOK_PLACE
ACTIVE BEFORE INSERT POSITION 0
as
begin
  if (new.UNIKEY is NULL) then
new.UNIKEY=GEN_ID(sysnumber,1);
  if (new.PLACEKEY is NULL) then exception
NO_PLACEKEY;
  if (new.BOOKKEY is NULL) then exception NO_BOOKKEY;
  if (NOT EXISTS(select * from TPLACE where
(UNIKEY=new.PLACEKEY)))
   then exception NO_PLACEKEY;
  if (NOT EXISTS(select * from TBOOK where
(UNIKEY=new.BOOKKEY)))
   then exception NO_BOOKKEY;
end
л

CREATE TRIGGER I_TBOOK_READER_1 FOR TBOOK_READER
ACTIVE BEFORE INSERT POSITION 0
as
 declare variable num smallint;
begin
  if (new.UNIKEY is NULL) then
new.UNIKEY=GEN_ID(sysnumber,1);
  if (new.READER is NULL) then exception
NO_READERKEY;
  if (new.BOOKKEY is NULL) then exception NO_BOOKKEY;
  if (NOT EXISTS(select * from TREADER where
(UNIKEY=new.READER)))
   then exception NO_READERKEY;
  if (NOT EXISTS(select * from TBOOK where
(UNIKEY=new.BOOKKEY)))
   then exception NO_BOOKKEY;
```

```
  select num_presence from TBOOK where
 (UNIKEY=new.BOOKKEY) into :num;
  if(num < 1) then exception NO_PRESENCE;
  update TBOOK set num_presence=:num;
end


CREATE TRIGGER I_TPLACE_1 FOR TPLACE
ACTIVE BEFORE INSERT POSITION 0
as
begin
  if (new.UNIKEY is NULL) then
new.UNIKEY=GEN_ID(sysnumber,1);
  if (new.CDPLACE is NULL) then exception NO_PLACE;
  if (new.TXPLACE is NULL) then exception NO_PLACENM;
end


CREATE TRIGGER I_TREADER_1 FOR TREADER
ACTIVE BEFORE INSERT POSITION 0
as
begin
  if (new.UNIKEY is NULL) then
new.UNIKEY=GEN_ID(sysnumber,1);
  if (new.RDNUMB is NULL) then exception NO_RDNUMB;
  if (new.RDNAME is NULL) then exception NO_RDNAME;
end


CREATE TRIGGER NORUBRICS_BU FOR NORUBRICS
ACTIVE BEFORE UPDATE POSITION 0
AS
BEGIN
  if((new.unikey is NULL or new.unikey=old.unikey)
   and (new.b1 is NULL or new.bl=old.bl)) then
    update tbook set tbook.BOOKNM=new.booknm
    where tbook.unikey=old.unikey;
END
л


SET TERM ; ^
CREATE ROLE ROLE1;
```

Дополнительная литература

1. Архангельский А.Я. Язык SQL в Delphi 5.0. - М.: Бином, 2000. - 208 с.
2. Архангельский А.Я. Язык SQL в C++Builder 5. (Серия: Все о C++Builder) - М.: Бином, 2000. - 208 с.
3. Баженова И.Ю. Oracle 8.8i. Уроки программирования. - М.:Диалог-МИФИ, 2000. - 304 стр.
4. Бобровский С. Delphi 5 учебный курс. СПб: Питер, 2000. - 640 с.
5. Боуман Д, Эмерсон С, Дарновски М. Практическое руководство по SQL. - Киев: Диалектика, 1997.
6. Вильям Дж. Пэйдж (мл.). Использование Oracle 8/8i. Специальное издание. - Изд. Вильямс, 1999.
7. Грабер М. Введение в SQL. - М.: Лори, 1996. - 379 с.
8. Грабер М. Справочное руководство по SQL. - М.: Лори, 1997. - 291 с.
9. Грачев А.Ю. Введение в СУБД. - М: Диалог-МИФИ, 2000. - 272 с.
10. Грин Д. "Oracle8 Server. Энциклопедия пользователя". - Изд. ДиаСофт, 2000.
11. Дейт К. Введение в системы баз данных - 6-е издание. - Изд.Вильямс, 1999. - 848 с.
12. Джексон Г. Проектирование реляционных баз данных для использования с микроЭВМ. - М.: Мир, 1991. - 252 с.
13. Епанешников А. М., Епанешников В. А. Delphi 5. Базы данных. - М: Диалог-МИФИ, 2000.
14. Ершов Л.Е., Палютин Е.А. Математическая логика. - М. Наука, 1979. - 320 с.
15. Коннолли Т., Бегг К., Страчан А. Базы данных: проектирование, реализация и сопровождение. Теория и практика - М: Диалектика, 2000. - 1120 с.
16. Кузнецов С.Д. SQL Язык реляционных баз данных. - Изд. Майор, 2001. - 192 с.
17. Ладани Ханс. SQL. Энциклопедия пользователя. - Изд. ДиаСофт, 1998.
18. Маклаков С В. Bpwin и Erwin CASE-средства разработки. информационных систем. - М.: Диалог-МИФИ, 1999. - 256 с.
19. Мамаев Е., Вишневский А. Microsoft SQL Server 7 для профессионалов. - СПб: Питер, 2000.

20. Мартин Д. Организация баз данных в вычислительных системах. - М.: Мир, 1980. - 662 с.

21. Пушников А.Ю. Введение в системы управления базами данных. Часть 1. Реляционная модель данных: Учебное пособие/Изд-е Башкирского ун-та. –Уфа, 1999. - 108 с.

22. Рейнсдорф К., Хендерсон К. Borland C++ Builder. Освой самостоятельно. - М.: Бином, 1998. – 704с.

23. Тихомиров Ю.В. Microsoft SQL Server 7.0. Разработка приложений. - СПб: BHV – 1999. – 352с.

24. Ульман Д. Основы систем баз данных. - М.: Финансы и статистика, 1983. -334 с.

25. Фаронов В.В. Delphi 5 учебный курс. - Изд. Нолидж, 2000. - 608 с

26. Харрингтон Д.Л. Проектирование реляционных баз данных. Просто и доступно. - М.: Лори, 2000.

27. Хичкок Б. SYBASE настольная книга администратора. - М: Лори, 2000. - 420 стр.

28. Хомоненко А. Д., Цыганков В. М., Мальцев М. Г. Базы данных. - Изд. Корона Принт, 2000

29. Codd E.F. Relation Model of Data for Large Shared Data Banks //Comm. ACM. - 1970. - V.13, №.6. - P.377-383. (Имеется перевод: Кодд Е.Ф. Реляционная модель данных для больших совместно используемых банков данных //СУБД. - 1995. – №1. – С.145–160.)

30. Codd E.F. Normalized Data Base Structure: A Brief Tutorial //Proc. of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control. - N.-Y.: ACM. - 1971. – P.1-17.

ОГЛАВЛЕНИЕ