ПРОГРАММА КУРСА “РАБОТА С БАЗАМИ ДАННЫХ (INTERBASE 6.X И DELPHI 5.0)”

Содержание

Глава 1.1 Разработка баз данных

В данной главе описываются общая методика построения баз данных применительно к Interbase, при этом я не претендую на все-объемлющий охват принципов разработки таких баз.

Эта глава включает в себя:

– обзор основных принципов и целей построения баз данных;
– основные этапы построения базы данных;
– специфика построения баз данных применительно к Interbase;
– предложения по планированию безопасности базы данных.

§ 1.1.1 Предварительный обзор

Основой для развития баз данных и их теоретических разработок является все более возрастающие потребности современного общества в информации и ее анализе. Компьютеры представляет собой удобный и естественный инструмент, позволяющий автоматизировать процессы обработки информации. Традиционно к базам данных относятся те области обработки информации, которые связанны с долговременным хранением в некотором стандартизированном виде достаточно большого объема взаимосвязанной информации с возможностью поиска и анализа данных. Естественно, системы баз данных могут объединяться с другими видами систем обработки информации, такими как расчетно-научные задачи, системы визуализации данных различных уровней, системы управления технологическими процессами и т. д., однако базы данных отличаются от прочих систем вышеуказанными свойствами:

– договременное хранение информации;
– стандартизированные способы описания хранимой информации;
– возможностью поиска информации по определенным критериям;

Свойство долговременного хранения достаточно большого объема информации характерно не только для систем баз данных, но и, например, для файловых систем. Однако файловые системы не предоставляют других возможностей, присущих базам данных, в связи с чем использование файловых систем ограниченно либо обработкой и хранением небольших объемов данных, либо достаточно больших объемов не связанных или слабосвязанных данных. В случае же необходимости выполнять более сложные манипуляции с большим объемом сильносвязанной информации необходимо использовать базы данных.

Базы данных (БД) представляют реальный мир, отображая его объекты в символическом виде как таблицы и другие объекты. После того, как соответствующая информация о реальном мире систематизирована и запомнена в базе данных, пользователи могут получить к ней доступ через соответствующие приложения на рабочих станция или терминалах.

Наиболее важным фактором при создании хорошо работающих баз данных является их качественное проектирование. Проектирование баз данных – это многоэтапный повторяющийся процесс, состоящий в декомпозиции сложных гетерогенных структур в набор взаимосвязанных небольших однородных объектов. Такой процесс называется нормализацией. Целью нормализации является получение определения естественных взаимосвязей между данными в базе. Это в основном достигается путем разделения одной таблицы на несколько таблиц с меньшим количеством полей. Когда одна таблица разбивается на несколько в процессе нормализации, данные не теряются, так как исходная таблица может быть получена путем операции соединения нормализованных таблиц. Упрощение таблиц такого рода приводит к тому, что в одной таблице оказываются собранными наиболее близкие данные и свойства реальных объектов.

База данных и модель данных.

Весьма важно провести различие между описанием базы данных (ее моделью) и собственно базой данных. Модель данных создается во время разработки базы и является шаблоном для создания таблиц и полей. Модель создается раньше того, как появится таблица или какие-либо связанные с ней данные в базе. Модель данных описывает логическую структуру базы данных, включая сюда объекты базы данных, типы данных, пользовательские операции, связи между объектами и ограничения целостности данных.

В реляционных базах данных описание базы полностью отделено от ее физической реализации, что придает процессу разработке больше гибкости:

- нет необходимости описания физического размещения объектов базы данных во время ее разработки, что позволяет уточнять и изменять практически любые взаимосвязи;
- логическая структура базы данных не зависит от изменений на физическом уровне. Это позволяет добиться межплатформенной совместимости. В результате можно легко переносить базу данных на другие платформы, так как механизм доступа к данным, описываемый моделью данных, не зависит от того, как хранятся данные.
- логическая структура данных также не зависит от представления данных для конечных пользователей. Разработчик может создать специальное представление таблиц баз данных, так называемое отображение (view, см. § 1.3.10). Отображение показывает только часть данных какому-либо определенному пользователю или группе. Отображения

могут использоваться для сокрытия засекреченных данных или для фильтрации данных, не интересных пользователю.

Цели разработки базы данных

Хотя реляционные базы данных обладают большой гибкостью, единственным путем обеспечения целостности данных и удовлетворительной производительности базы данных является ее тщательная разработка – в реляционных базах нет встроенной защиты от логических ошибок, сделанных во время разработки. Качественно разработанная база данных должна отвечать следующим условиям:

- удовлетворять запросам пользователей; перед тем, как создать базу, необходимо хорошо исследовать потребности пользователей и выяснить, как и для чего база будет использоваться;
- обеспечивать правила модели данных, логичность и целостность данных: во время разработки таблиц необходимо определять свойства и ограничения, которые лимитировали бы данные, вносимые в базу пользователем или программой путем их проверки до записи в таблицу(ы) базы данных;
- обеспечивать легкую и понятную структуру информации: хорошо разработанная база позволяет проще понимать запросы и избегать ввода пользователями противоречивой или повторяющейся информации. Это облегчает модернизацию и поддержку базы данных.
- удовлетворять требованиям производительности; хорошо разработанная база обеспечивает высокую производительность: когда таблицы слишком велики (по количеству столбцов) или когда индексов слишком мало (или наоборот, слишком много), это приводит к длительным задержкам при работе с базой данных; кроме того, если база данных очень велика и выполнятся большое количество транзакций, то производительность такой базы очень сильно зависит от качества ее разработки.

§ 1.1.2 Основные этапы разработки базы данных

1. Определение информационных требований к будущей базе данных путем опроса пользователей
2. Анализ реальных объектов предметной области, которую необходимо моделировать в базе данных. Организация объектов по категориям и свойствам и создание их списка.

3. Назначение соответствия категорий и свойств таблицам и их полям.
4. Определение свойств, которые однозначно определяют каждый объект.
5. Разработка набора правил, который определяет, каким образом можно обращаться, добавлять и модифицировать данные в конкретных таблицах.
6. Установление соотношения между объектами (таблицами и полями).
7. Определение права доступа к базе данных

Ниже в этой главе все вышеприведенные этапы расписаны более подробно.

§ 1.1.3 Анализ требований к базе

Первым шагом процесса разработки базы данных является исследование предметной области, которую предстоит моделировать. Исследование основывается на опросе будущих пользователей базы данных для прояснения всех деталей модели и для формирования списка требуемых отчетов и форм. Вот типичный список вопросов (помимо вопросов о предметной области), который следует задавать пользователям:

1. Должны ли ранее разработанные приложения продолжать работать во время разработки новой базы данных? Должна ли новая база быть приспособлена к существующим приложениям или, возможно, необходимо переделать старые приложения, чтобы они гармонично вписывались в новую систему?
2. Как разные приложения используют общие данные? Будет ли новая база данных разделять с другими приложениями какие-либо данные?
3. Каким образом пользователи будут использовать данные, хранимые в базе? Кто будет вводить данные и в каком виде? Насколько часто будут меняться объекты базы данных?
4. Какой доступ требуется существующим приложениям? Необходимо ли вашей системе иметь доступ к одной базе данных или к нескольким разным базам, которые могут существенно отличаться по структуре? Какой сервис другие СУБД предлагают внешним приложениям и насколько сложно будет его реализовать в разрабатываемой БД?
5. Какая информация наиболее критична ко времени доступа, требует быстрый просмотр или изменение?

В соответствии с ответами на вышеозначенные вопросы должна прорабатываться структура базы данных с учетом необходимости поддержки старых приложений, интенсивности обновления базы данных, необходимость создания индексов и т. д.

§ 1.1.4 Сбор и анализ данных

Перед разработкой объектов базы данных (таблиц и полей), необходимо проанализировать реальные данные на концептуальном уровне. Основные цели данного этапа:

- Определить основные функции и возможные действия, присущие данной предметной области. Например, наем служащих, отгрузка продуктов, заказ деталей, осуществление платежей и т. д.
- Определить объекты, над которыми осуществляются действия или которые выполняют какие-либо функции. Разделение бизнес-операции или транзакции на несколько стадий может помочь в определении всех возможных сущностей и отношений между отдельными операциями. Например, когда Вы рассматриваете процесс “наем служащих”, Вы можете легко определить такие сущности как ДОЛЖНОСТЬ, СЛУЖАЩИЙ и ОТДЕЛ.
- Определение свойств этих объектов. Например, сущность СЛУЖАЩИЙ может включать в себя такие атрибуты, как ID (идентификатор служащего), ФАМИЛИЯ, ИМЯ, ОТЧЕСТВО, ДОЛЖНОСТЬ, ОКЛАД и т. д.
- Определение взаимоотношений между объектами. Например, как сущности ДОЛЖНОСТЬ, СЛУЖАЩИЙ и ОТДЕЛ соотносятся между собой? В простейшем случае служащий имеет одну должность и работает в одном отделе, в то время как в одном отделе может работать много служащих и иметься несколько должностей. Построение простейшего графического представления модели предметной области может помочь определить взаимосвязи между объектами.

§ 1.1.5 Определение сущностей и их свойств

На основе требований, которые были собраны на предварительном этапе, определяются объекты, необходимые для построения базы данных – сущности и их свойства.

Сущность – это лицо, внешний объект или вещь, которую необходимо описать в базе данных. Она может существовать физически, например, как человек, автомобиль или служащий, или иметь какой-

либо концептуальный смысл: компания, должность или проект. Сущность представляет множество объектов реального мира, обладающих одинаковым набором свойств. Такие свойства называются атрибутами. Например, Вы разрабатываете базу, в которой должна содержаться информация о каждом служащем компании, информация об отделах и текущих проектах и информация о клиентах и продажах. Ниже приведен пример организации списка сущностей и их атрибутов, соответствующих данным разрабатываемого проекта.

Сущность	Атрибуты	Сущность	Атрибуты
СЛУЖАЩИЙ	*Номер служащего*	ПРОЕКТ	*Идентификатор*
	Фамилия		*Наименование*
	Имя		*Описание проекта*
	Отчество		*Руководитель проекта*
	Номер отдела		*Исполнители*
	Должность		*Продукты*
	Оклад	КЛИЕНТ	*Номер*
	Телефон		*ФИО клиента*
			Контактное имя
ОТДЕЛ	*Номер отдела*		*Телефон*
	Наименование		*Адрес*
	ФИО начальника	ПРОДАЖИ	*Номер поставки*
			Номер клиента
	Бюджет		*Номер служащего*
	Расположение		
	Телефон		*Цена*
ПРОДУКТ	*Номер*		*ИД продукта*
	Описание		*Дата заказа*
	Стоимость		*Дата отгрузки*
	Розничная цена		*Состояние заказа*

Объединив сущности и атрибуты подобным образом, можно приступить к исключению лишних сущностей и атрибутов. Все ли сущности в Вашем списке можно представить таблицами? Возможно, некоторые атрибуты стоит переместить в другую группу? Не появляются ли одни и те же атрибуты у разных сущностей? Каждый атрибут должен появляться только единожды и Вам необходимо определить, которой из сущностей должен единолично принадлежать атрибут. Например, атрибут ФИО НАЧАЛЬНИКА должен быть удален из базы

данных, так как имена служащих уже присутствуют в сущности СЛУЖАЩИЙ. Вместо него необходимо добавить атрибут НОМЕР СЛУЖАЩЕГО, который будет использоваться для получения всей необходимой информации о начальнике отдела путем ссылки на атрибут НОМЕР сущности СЛУЖАЩИЙ. Более подробную информацию о получении информации по ссылке см. ниже.

§ 1.1.6 Графическое представление сущностей и связей между ними

Для графического представления информации о связях между сущностями при проектировании баз данных обычно используются так называемые ER-диаграммы (диаграммы “сущность-связь”). Они основаны на одноименной модели данных, разработаннщой в 1976 г. Ченом. Использование этой модели и диаграмм позволяет очень удобно и наглядно представить взаимосвязь между сущностями БД в целом и существенно упростить проектирование, поэтому такие диаграммы получили широкое распространение. При разработке баз данных рекомендуется документировать связи между сущностями именно таким образом, хотя имеются и другие способы графического представления модели данных, однако они не получили достаточного распространения.

Основные концепции ER-модели

ER-модель базируется на следующих концепциях: сущности, связи и атрибуты. Дадим определение этих концепций:

Сущность – некий реальный или абстрактный объект, имеющий в моделируемой предметной области независимое существование. В общем, это более формальное определение, чем данное в предыдущем параграфе, однако имеющее тот же смысл. Конкретное представление одного из моделируемых объектов будем называть экземпляром сущности. Однако не менее распространено и другое определение, где вводится понятие типа сущности и просто сущности, что соответствует сущности и экземпляру сущности данной работы.

Так же в ER модели сущности делятся на два класса:

- слабая сущность – сущность, существование которой зависит от какого-то другой сущности
- сильная сущность – ее существование, в отличие от слабой, не зависит от других сущностей.

Для предметной области обучение сильной сущностью будет предмет, а слабой сущностью – расписание, так как сущность расписание не может существовать без предметов, а так же, возможно, других сущностей. Обычно между сильной сущностью и слабой присутствует та или иная связь, которая должна отображаться на диаграмме.

Каждый сильная сущность отображается на диаграмме в виде прямоугольника с именем сущности внутри него, а слабая сущность – таким же образом за исключением того, что контур прямоугольника должен быть двойным. Пример отображения сущностей приведен на рис. ?.

Рис.

Атрибут – это свойство сущности или связи. Как видим, в концепции модели сущность-связь понятие атрибута расширяется, так атрибуты могут характеризовать не только сущности, но и связи между ними (см. далее). Кроме того, добавляется дополнительное разделение атрибутов в зависимости от характера представляемых атрибутом данных:

- **простой атрибут** – атрибут, состоящий из одного компонента с независимым существованием (т. е. его значение не зависит от значения каких-либо атрибутов данной или связанной с данной сущностями); простые атрибуты не могут быть заменены на более мелкие; например: пол человека, зарплата сотрудника, рост человека и т. д.
- **составной атрибут** – это атрибут, состоящий из нескольких компонентов, каждый из которых характеризуется независимым существованием. Составной атрибут всегда можно разбить на ряд более мелких простых атрибутов. Например, атрибут *ФИО* можно разбить на атрибуты *фамилия*, *имя* и *отчество*. Определение типа конкретного атрибута (простого или составного) в разрабатываемой системе зависит от пользовательского представления данного свойства – как единого целого или как набор отдельных компонентов. В частности, если пользовательский интерфейс предполагает поиск только по имени (или другому компоненту полного имени человека), то атрибут должен быть составным.
- **однозначный атрибут** – атрибут, содержащий только одно значение для каждого атрибута конкретного экземпляра сущности. В большинстве случаев атрибуты являются однозначными. Например, атрибут *ФИО начальника* экземпляра сущности *Отдел* всегда имеет единственное значение и, соответственно, является однозначным.
- **многозначный атрибут** – атрибут, который для одного экземпляра сущности может содержать несколько значений.

Например, атрибут *Телефон* сущности *Служащий* может содержать несколько значений и поэтому является многозначным.

- **производный атрибут** – значение такого атрибута зависит от значения базового атрибута или определенного множества атрибутов, принадлежащих какой-либо сущности (не обязательно данной). В качестве примера можно указать атрибут, хранящий сведения о ежемесячном подоходном налоге (если налог не является прогрессивным), который зависит от начисленной за каждый месяц заработной платы или возраст человека, который зависит от даты его рождения. Также бывают и более сложные виды производных атрибутов. Например, атрибут количества служащих в отделе может быть рассчитан как количество экземпляров сущности *Служащий*, у которых атрибут *Номер отдела* равен данному отделу.
- **первичный и возможные ключи** – атрибуты, однозначно выделяющие конкретный экземпляр данной сущности из всей их совокупности (так же см. § 1.1.8);

Атрибуты отображаются на диаграммах в виде эллипсов, при-

Рис.

соединенных линией к соответствующей сущности. Производный атрибут отображается эллипсом с пунктирной границей, а многозначный – двойной границей. Если атрибут является составным – то его атрибуты-компоненты отображаются в виде присоединенных к нему

эллипсов. Атрибут, являющийся первичным ключом сущности – подчеркивается. Возможные ключи сущности подчеркиваются пунктирной линией. Пример отображения атрибутов сущности приведен на рис. На этом рисунке рассмотрено представление сущности СЛУЖАЩИЙ, при этом атрибут НОМЕР является первичным ключом сущности, а атрибут ФИО – возможным. Так же атрибут ФИО является составным, что указывает на необходимость получать доступ к отдельным составляющим атрибута. Атрибут ТЕЛЕФОН является многозначным, так как у одного служащего может быть несколько телефонов. Атрибут СРЕДНИЙ ОКЛАД является производным, так как для его получения необходимов вычислить среднее значение для всех служащих.

Типы связей

Типом связи называется набор осмысленных ассоциаций между различным сущностями. Каждому типу связи дается название, соответствующее выполняемой функции. Следует обратить внимание на то, что имеется понятие тип связи и связь, которые необходимо различать. **Связь** – это такая ассоциация между сущностями, которая включает в себя по одному или нескольких экземпляров из каждой участвующей в связи сущности.

Связь характеризуется степенью и кардинальностью и участием. **Степень связи** определяется количество сущностей, задействованных в ней (могут быть бинарные, тернарные, квартернарные и более высоких степеней; иногда рекурсивную связь называют унарной см. ниже). **Показатель кардинальности** – определяет количество экземляров сущностей на каждой стороне связи (фактически этот показатель, как и следующий, является ограничением накладывающимся на связь и относится к так называемым бизнес-правилам; хотя не все бизнес-правила укладываются в показатель кардинальности). Наиболее распространными являются бинарные связи с показателем кардинальности один к одному (1:1), один ко многим (1:m) и многие ко многим (m:n) (см. рис. ,также § 1.1.9). **Степень участия** показывает, зависит ли существование какого-либо экземпляра сущности от участвующих в связи экземпляров других сущностей. Может быть полной или частичной. Степень участия является полной, если для существования экземпляра данной сущности обязательно наличие

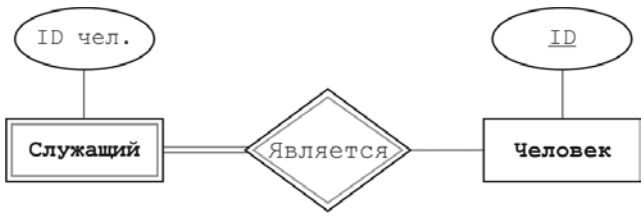

Рис.

экземпляра другой сущности. Допускается и более гибкое указание степени участия путем задания минимального и максимального количества экземпляров сущностей, участвующих в связи.

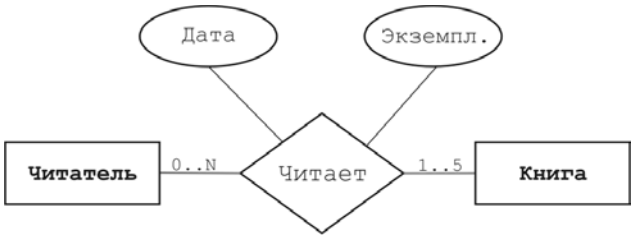

Рис.

Каждая связь может иметь атрибуты, присущие только связи, но не сущностям, входящим в эту связь. В качестве примера можно привести связь (см. рис.) между сущностями *Книга* и *Читатель* базы данных библиотеки. В случае взятия книги читателем необходимо хранить информацию о дате и количестве экземпляров. Эти данные скорее относятся к связи, чем к одной из сущностей. Обычно наличие таких атрибутов у связи говорит о том, что присутствует некоторая скрытая сущность, не учтенная на данном этапе проектирования.

Рис.

Для представления связи на диаграммах используется ромбик с указанием имени связи. Если связь используется для установления отношения между сильной и слабой сущностями, то используется двойной ромбик. Так же возможно указание уточняющих надписей возле каждой сущности, участвующей в связи, это особенно актуально при отображении так называемых рекурсивных отношений (когда один экземпляр сущности ссылается на другой экземпляр этой же сущности, при этом они играют разные роли, рис.). Полное и неполное участие отображается двойной или одинарной линией соответсвенно (см. рис. .). Показатель кардинальности подписывается на линиях, соединяющих связь с сущностями в виде целых чисел или диапазона (минимальное количество экземляров сущности и максимальное), что дает более полное представление о ограничениях данной связи (рис.).

Рассмотрим построение ER-модели (рис.) на примере, рассмотренном в предыдущем параграфе. При этом не забудем, что допускается отображать только основные атрибуты, необходимые для понимания модели данных, а второстепенные опускать, что улучшает восприятие диаграммы.

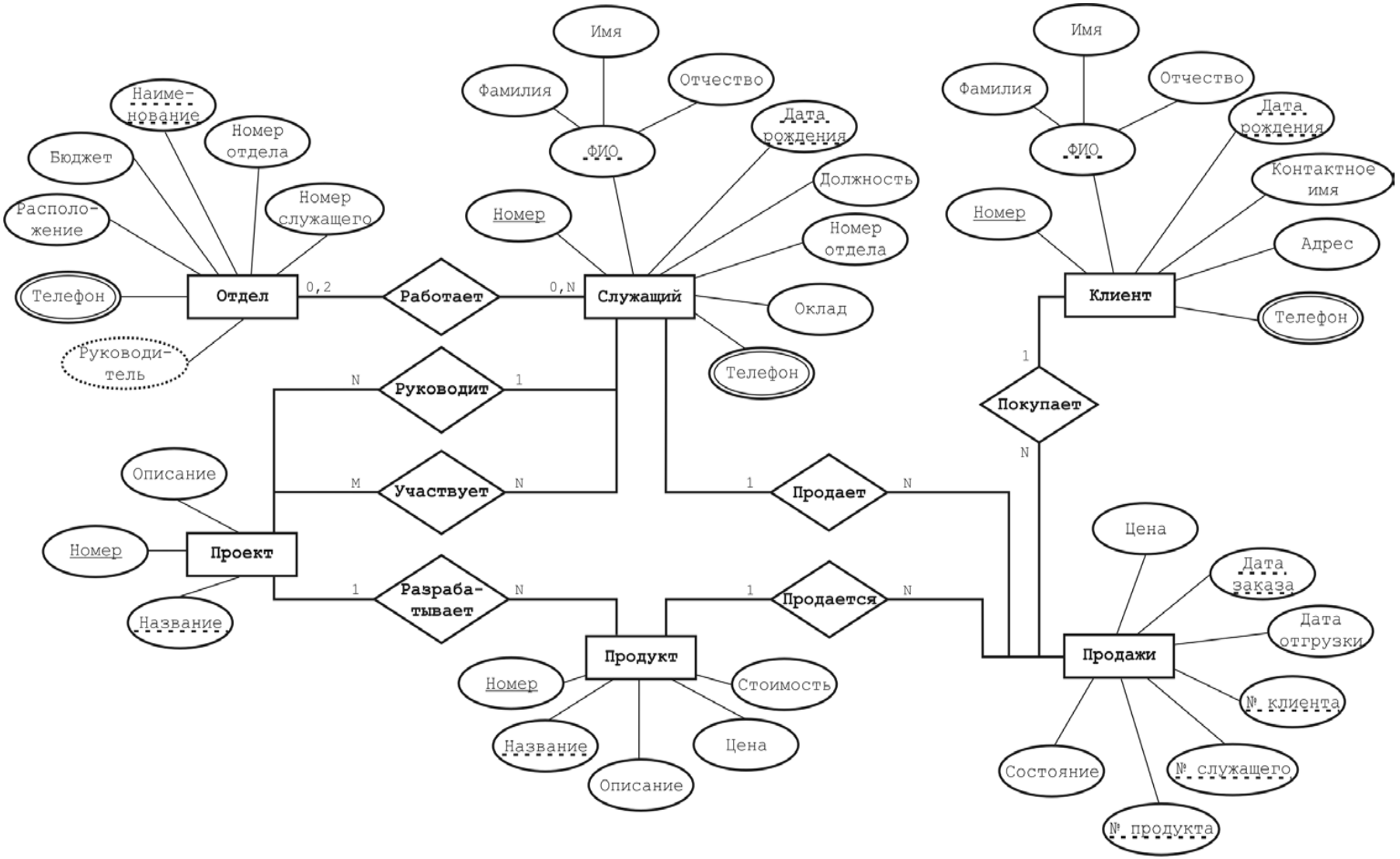
Имя
Фамилия
Отчество
ФИО
Дата рождения
Должность
Номер отдела
Оклад
Телефон
Номер
Служащий
Работает
0,2
0,N
Отдел
Наиме-нование
Номер отдела
Бюджет
Номер служащего
Распо-ложение
Телефон
Руководи-тель
Руководит
N
1
Участвует
M
N
Описание
Проект
Номер
Название
Разраба-тывает
1
N
Продает
1
N
Имя
Фамилия
Отчество
ФИО
Дата рождения
Контактное имя
Номер
Адрес
Клиент
Телефон
Покупает
1
N
Цена
Дата заказа
Дата отгрузки
№ клиента
№ служащего
№ продукта
Состояние
Продажи
Продается
1
N
Продукт
Номер
Стоимость
Название
Цена
Описание

Рис.

Следующий же параграф описывает, каким образом отобразить сущности и атрибуты в конкретные объекты базы данных – сущности в таблицы, а атрибуты в колонки таблиц. Дополнительные сведения о правилах описания сущностей, атрибутов и связей между ними можно получить, изучив теорию нормальных форм (Глава 1.2)

§ 1.1.7 Разработка таблиц

В реляционной базе данных каждая сущность представляется таблицей. Таблица – это двумерная матрица, состоящая из строк и столбцов. Каждый столбец представляет собой атрибут сущности, а строка – конкретное ее воплощение. После определения сущностей и атрибутов предметной области, необходимо создать модель данных, которая послужит основой для построения базы данных. Модель данных отображает сущности и атрибуты в таблицы базы данных и детально описывает используемые колонки таблиц и отношения между таблицами.

Пример ниже показывает, как сущность СЛУЖАЩИЙ может быть преобразована в таблицу:

Таблица 1

ID	First_Name	Patronymic_Name	Last_Name	Dept_Code
1	Иван	Васильевич	Ковалевский	25
2	Петр	Иванович	Митрофанов	25
3	Анна	Викторовна	Криволапова	12

Продолжение таблицы 1

Job	Salary	Phone
инженер	2000.00р	267-34-21
ведущий инженер	3000.00р	223-89-90
оператор	1200.00р	934-12-67

Так как при создании на сервере InterBase (а так же на большинстве других серверов) таблиц с русскими именами обычно возникают проблемы, то колонки таблицы названы по-английски; в таблице 2 приведено соответствие между названиями атрибутов и названиям колонок таблицы:

Каждая строка в таблице 1 представляет одного служащего. Колонки ID, First_Name, Patronymic_Name, Last_Name, Dept_Code, Job, Salary и Phone представляют соответствующие атрибуты сущности СЛУЖАЩИЙ в соответствии с таблицей 2. Когда таблица заполняется данными, в нее добавляются строчки, а значение, хранимое на пересечении строки и столбца, называется полем.

Таблица 2

Атрибут	Название колонки
Номер служащего	ID
Фамилия	Last_Name
Имя	First_Name
Отчество	Patronymic_Name
Номер отдела	Dept_Code
Должность	Job
Оклад	Salary
Телефон	Phone

§ 1.1.8 Определение уникальных атрибутов

Так как реляционные базы данных всегда работают с множествами, к элементам которых нельзя обратиться по номеру или местоположению, то одной из задач разработки базы данных является определение способа однозначной идентификации каждого экземпляра сущности по имеющимся у него атрибутам. Так же наличие ограничений уникальности позволяет избежать ввода неверных (повторяющихся) данных. Элемент данных, который позволяет однозначно идентифицировать экземпляр сущности, называется **ключом**. Для однозначной идентификации может подходить одновременно несколько атрибутов сущности или их сочетаний, они носят название **возможных ключей**. Тот из них, который выбран в модели базы данных для идентификации конкретного экземпляра сущности, называется **первичным ключом**. Если ключ состоит из нескольких полей, то он называется **составным**.

Значение, определенное в первичном ключе таблицы, является признаком, однозначно определяющим каждую запись. Соответствующие конструкции языка SQL, PRIMARY KEY или UNIQUE требуют, что бы данные, вводимые в поле (или несколько полей), были уникальными для каждой строки соответствующей таблицы. Если пользователей попытается ввести данные в поле (или их набор), имеющей свойства первичного ключа или уникальности, которые уже имеются в другой записи, сервер запретит такую операцию и вернет сообщение об ошибке.

Например, в таблице СЛУЖАЩИЙ, поле ID может служить первичным ключом, так как оно по своей сути уникально для каждого работающего на предприятии служащего. Когда разработчик решает задачу выбора первичного ключа, он должен основываться на том, какой атрибут (или их совокупность) являются уникальными по своей

сути. Например, не может быть двух одинаковых номеров паспорта или водительских удостоверений, поэтому их можно использовать в качестве первичных ключей. С другой стороны, имя человека может быть одинаковым у разных людей, в силу этого этот атрибут не является уникальным и поэтому его нельзя использовать в качестве первичного ключа. В случае, если ни один атрибут не может являться первичным ключом, можно использоваться набор атрибутов для создания составного первичного ключа, однако набор атрибутов также должен быть естественно уникален. В частности, набор ФАМИЛИЯ, ИМЯ, ОТЧЕСТВО плохой пример первичного ключа, так как достаточно часто встречаются люди, у которых эти атрибуты совпадают.

Для связи двух таблиц составные ключи подходят не очень хорошо, так как в этом случае в каждой связанной таблице хранятся дублированные значения первичного ключа, поэтому место в базе данных расходуется не оптимально и страдает производительность. Для выполнения этой задачи рекомендуется вводить искусственное поле в виде целого числа-идентификатора, которое будет являться ключевым. При этом желательно в таблицах кроме искусственного ключа иметь один или несколько (в зависимости от особенностей моделируемой сущности) естественных ключей. Это необходимо как с точки зрения логики модели, так и с точки зрения теории реляционных баз данных, в которых все операции осуществляются над множествами записей, поэтому желательно иметь возможность с помощью какого-либо простого или сложного условия выделить отдельную запись. Кроме того это существенно облегчает процесс разработки клиентских приложений, когда поле-идентификатор автоматически генерируется сервером. Возможные отклонения от этого правила связаны с вопросами производительности базы данных, так как каждый дополнительный ключ снижает производительность модификации записей.

Отличие первичного ключа от уникального (вторичного) ключа состоит только в том, что (обычно) уникальный ключ не служит целям идентификации записей и не используется для установки связей между таблицами. Уникальный ключ требуется для того, что бы гарантировать уникальность значений в поле (или их наборе) во всей таблице, кроме того, для уникального ключа создается отдельный индекс, хотя он вполне может выполнять все функции, присущие первичному ключу. При создании таблицы можно описать один первичный ключ и практически неограниченное количество уникальных ключей.

§ 1.1.9 Определение основных правил базы данных

При разработке таблицы необходимо выработать набор правил для каждой таблицы и колонки, которые обеспечивают целостность данных. Набор правил включает в себя:

- определение типа данных;
- выбор (при необходимости) набора символов;
- создание полей на базе домена;
- установление значений по умолчанию и определения необходимости заполнения поля;
- определения проверок данных (ограничения CHECK);
- определения ссылочной целостности и правил обновления;

Определение типа данных. После того, как определенный атрибут выбран в качестве колонки таблицы, необходимо определить его тип данных. Тип данных описывает набор допустимых значений, который может содержаться в данном поле, а также какие действия может производить с данными и сколько они будут занимать места.

Основные типы данных SQL:

- символьные типы данных (для хранения текстовых строк);
- целочисленные типы данных;
- типы данных с фиксированной и плавающей точкой;
- типы даты и времени;
- типы данных BLOB для представления данных неизвестной длины и структуры, например, изображений и оцифрованных звуков; BLOB могут быть цифровыми, текстовыми и двоичными;

Для более подробного ознакомления с типами данных см.§ 1.3.5.

Выбор набора символов. При создании базы данных можно определить набор символов, который в дальнейшем будет определять:

- какие символы могут быть использованы в текстовых полях;
- порядок, в котором такие поля будут сортироваться;
- процедуру сравнения двух текстовых значений.

Установка при создании базы данных набора символов по умолчанию особенно интересна разработчикам, создающим базы данных для международного использования. Для каждой колонки текстового типа возможно определение своего набора символов. См. также § 1.3.4

Определение доменов. В случае, если в базе встречается много полей с одними и теми же типами данных и ограничениями, можно описать домен и затем использовать его при создании таблиц, что может существенно сэкономить время разработки. Поля будут наследовать все свойства домена, при этом некоторые могут быть изменены при описании поля. Единственное ограничение, которое возникает при описании домена, связано с невозможностью определения для любого домена ограничений целостности. Это связано с тем, что на этапе создания домена структура базы данных может быть неизвестной или измениться в дальнейшем, т. е. домен – это абстракция, не связанная со свойствами сущностей, а только определяющая свойства атрибута безотносительно сущности, к которой этот атрибут относится.

Кроме того, домены нельзя представить и как пользовательские типы данных, характерных для языков программирования 3-го уровня, так как не допускается использование доменов (то есть домен может иметь только один из встроенных типов данных; нельзя описать домен на основе другого ранее описанного домена), и отсутствуют механизмы какой-либо проверки типов доменов.

Подробное описание команды создания доменов приведено в § 1.3.5

Значения по умолчанию и необходимость заполнения поля. При создании поля можно определить для него значения по умолчанию. В этом случае при создании новой записи, если значение для соответствующего поля было не определено, то в него будет записано значение по умолчанию. Значение по умолчанию сокращает время ввода данных и предотвращает возможные ошибки ввода. Например, значение по умолчанию для поля типа DATE может быть текущая дата, а в поле типа “Да/Нет” при ответе на какой-нибудь вопрос – значение “Да”. Значение по умолчанию для поля при вставке данных имеет больший приоритет по сравнению со значением по умолчанию для домена.

Для тех полей, в которые ввод данных не обязателен, устанавливается свойство NULL (пусто), которое разрешает оставлять их пустыми. Для полей, в которые ввод данных обязателен, устанавливается свойство NOT NULL (не пусто), которое требует либо чтобы данные были введены, либо чтобы было описано значение по умолчанию. Колонки, входящие в состав первичного или уникального ключа должны иметь свойство NOT NULL.

Определение проверок данных. Кроме ограничений уникальности, определяемых первичным и уникальным(и) ключами, возможно иное описание целостности данных. Проверка данных (CHECK) определяет условия или требования к данным в колонке (или строке) во время добавления (или изменения) данных в таблицу. Проверка – это условие, которое должно быть истинным для того, чтобы новые данные были внесены в таблицу.

Определение взаимосвязи между таблицами базы данных.

При разработке базы данных также необходимо определить взаимосвязи между таблицами. Например, как соотносятся между собой сущности служащие и отделы? Служащий обычно может работать только в одном отделе (отношение один к одному), однако в отделе может трудиться много служащих (отношение один ко многим). Как соотносятся между собой сущности служащие и проекты? Служащий может одновременно работать над разными проектами, а в одном проекте могут участвовать несколько сотрудников (такое отношение называется многие ко многим). Все вышеуказанные взаимосвязи должны быть описаны при разработке базы данных.

Отношение один к одному может быть реализовано путем объединения двух таблиц (представляющих различные сущности) в одну, либо, если по каким-либо причинам такое объединение невозможно, то такая связь может быть реализована с помощью отношения один ко многим (возможно, со специальными дополнительными ограничениями средствами базы данных). В силу простоты далее не рассматривается.

Отношение один ко многим реализуется встроенными средствами базы данных с помощью первичных (или уникальных) и так называемых внешних ключей. Рассмотрим организацию таких отношений на примере из § 1.1.5. Сущности служащие и отделы связаны между собой по номеру отдела. Для сущности *отдел* атрибут *номер отдела* является признаком, однозначно определяющим все остальные атрибуты, поэтому должен быть первичным (или, в некоторых случаях, уникальным) ключом. Атрибут *номер отдела* сущности служащие служит для указания (связи) с конкретным воплощением информации об отделе, т. е. в данном случае имеет место рассматриваемое в этом абзаце отношение один ко многим. При реализации базы данных для поддержания такого отношения необходимо построить по полю *номер отдела* таблицы со списком служащих внешний ключ, в котором указано значениям какого из полей другой (внешней, отсюда и название ключа) таблицы должны соответствовать значения, вводимые в дан

ное поле. После этого невозможно будет для служащего указать несуществующий в данный момент в базе номер отдела.

Для реализации отношения многие ко многим в базе данных создается дополнительная таблица, которая связывает двумя отношениями один ко многим две таблицы, связанные отношением многие ко многим. Например, для реализации связи между проектами и служащими создается дополнительная таблица, состоящая из двух полей: идентификатор служащего и идентификатор проекта, каждое из которых связано отношением один ко многим с соответствующей таблицей. Это позволяет вводить список служащих, работающих над одним проектом и одновременно иметь информацию о том, над какими проектами работает какой-либо служащий. То есть с помощью создания дополнительной таблицы мы реализовали отношение многие ко многим.

Индексы

После создания таблиц, описывающих сущности модели реальной предметной области, необходимо определиться с набором индексов. Индексы в большинстве случаев существенно ускоряют процесс выборки данных, однако замедляют добавление и изменение записей таблиц. Следует помнить, что для полей, имеющих свойства первичного или уникального ключей, или используемых во внешних ключах, индексы создаются автоматически, поэтому отдельно создавать индексы для таких полей не нужно. Также индексы занимают дополнительное место в базе данных.

Индексы создаются на базе одного или нескольких полей. При операции выборки данных из таблицы с условиями отбора по полям, имеющим индексы, или требующей сортировки по таким полям, время выполнения операции существенно сокращается, так как поиск или сортировка по индексу выполняется существенно быстрее из-за более оптимальных алгоритмов. Поэтому индексы требуется создавать для тех полей, по которым часто осуществляется поиск или сортировка.

Эффективность индексов существенно снижается, если в таблице хранится не слишком много записей. В частности, не стоит создавать индексы, если содержимое таблицы полностью помещается на 1–5 страницах базы данных. При создании базы данных можно определиться с размером страницы исходя из тех соображений, что на страницу базы должна полностью помещаться одна самая большая запись. При этом следует учитывать, что сервер InterBase поддерживает переменный размер записи и автоматически сжимает ее, если в ней имеются пустые поля или числовые поля со значением 0. В этом случае рекомендуется считать, что средний размер записи на 20% меньше

максимального. Если же таких полей нет или их ожидается очень мало, то рекомендуемый средний размер необходимо увеличить на 5% от максимального.

В процессе работы при ухудшении производительности необходимо проверять количество уровней индекса. Если количество уровней превышает 4, то необходимо увеличить размер страницы. Более подробно см. Глава 1.3.

Индексы малоэффективны и в том случае, если в поле хранится небольшое множество значений, например, если в таблице хранится 1000000 записей и какое-либо поле имеет значение 0 или 1, то использование индекса по этому полю будет неэффективным.

Не следует создавать индексы для таблиц, в которые очень часто должно происходит добавление записей в условиях, приближенных к реальному времени.

Кроме того, настоятельно не рекомендуется создавать несколько индексов, построенных по одному полю. Лучше построить три индекса для трех полей, чем все возможные их сочетания, так как в большинстве случаев InterBase будет достаточно оптимально использовать индексы, построенные на одиночных полях и для запросов с отбором или сортировкой по нескольким индексированным полям.

При тестировании производительности для выяснения необходимости создания индексов необходимо иметь в таблицах количество записей, сравнимое с тем, которое будет при реальной работы базы, т. е. не стоит проводить тестирование на 10 или 100 записях, если при реальной работе их будет на 2-3 порядка больше.

§ 1.1.10 Планирование безопасности

Это очень важный вопрос. Поэтому при создании базы данных разработчик должен найти ответы на следующие вопросы:

- кто имеет право пользоваться сервером InterBase?
- кто имеет право пользоваться конкретной базой данных?
- кто и какие имеет права доступа к объектам, хранящимся в конкретной базе данных?

Более подробную информацию касательно системы безопасности InterBase см. § 1.3.15

Глава 1.2 Теория нормальных форм

После того, как Вы создали описание всех таблиц базы данных, необходимо взглянуть на совокупность созданных объектов в целом и провести анализ с помощью теории нормальных форм для выявления возможных логических ошибок. Модель базы данных содержит как

структурную (связанную с отношениями между сущностями), так и семантическую (функциональные зависимости между атрибутами сущностей) информацию. Однако некоторые функциональные зависимости могут быть нежелательными в силу наличия побочных эффектов или аномалий обновления данных. В этих случаях необходимо прибегнуть к процедуре декомпозиции (разложения), при которой существующее множество отношений между сущностями изменяется и их становится больше. Целью декомпозиции является устранение нежелательных функциональных зависимостей (и, соответственно, аномалий поведения). Выполнение нескольких шагов декомпозиций и является сущностью процесса нормализации. В общем процесс нормализации, как уже было отмечено выше, сводится к разбиению больших исходных таблиц на некоторое количество более мелких для объединения в группы по их естественным особенностям, при этом новые отношение между таблицами имеют более простую и регулярную структуру. При этом процесс нормализации является обратимым, т. е. в результате декомпозиции мы должны гарантировать отсутствие потерь в схеме данных и сохранить уже имеющиеся зависимости между сущностями.

При описании теории нормальных форм пользуются следующими определениями:

- атрибут, входящий в ключ, называется первичным, иначе он называется непервичным;
- зависимость $A \rightarrow B$ называется полной, если B зависит от всей группы атрибутов A, а не от ее подмножества. Например, если $A = A_1, A_2, \ldots, A_n$ и $A_1, A_2 \rightarrow B$, то такая зависимость будет неполной.

В данной главе я не буду давать детального описания теории нормальных форм (оно имеется в следующих источниках:), а ограничусь только практически значимыми определениями и советами.

После окончания нормализации однотипные данные хранятся в базе только в одном месте. Это приводит к следующим преимуществам:

- данные легче модифицировать и удалять;
- когда данные хранятся в одном месте и обращение к ним происходит по ссылкам, снижается вероятность ошибок, связанных с наличием дублирующих записей;
- снижается вероятность внесения несогласованных данных;
- уменьшается требования к объему памяти, занимаемой базой.

Однако нормализация приводит к некоторым недостаткам:

– скорость выборки данных может снижаться, иногда весьма значительно;
– добавление данных существенно усложняется и замедляется.

Конкретные рекомендации, в каких случаях использовать нормализацию, в каких нет, и если использовать, то до какого уровня, приведены в конце этой главы.

Процесс нормализации обычно состоит из следующих этапов:

– удаление повторяющихся групп;
– удаление частично-зависимых полей;
– удаление транзитивно-зависимых (косвенно-зависимых) полей.

Далее подробно описывается каждый этап нормализации, который связан с выполнением для каждой исходной таблицы опрделенных правил, называемых нормальными формами.

Первая нормальная форма (1НФ).

Таблица находится в первой нормальной форме в том случае, если ни одно из полей не содержит множеств значений или в таблице не представлены данные в иерархическом виде или повторяющейся группой. Например, следующие наборы записей не находятся в первой нормальной форме:

Пример 1

ФИО	**Дети**
Иванов И. И.	Сергей, Анна
Петров С. В.	Елена
Сидоров Е. А.	Анна, Мария, Ксения
Козлов Б. Е.	Владимир
Попугаев Д. М.	Виктор, Наталья, Ольга

Пример 2

Преподаватель	**Курсы**
Валентинов Д. Е.	Высшая математика
	Дискретная математика
	Математическая логика
	Теория множеств
Блюмберг Р. Э.	Информатика
	Прикладное программирование
	Мультимедия технологии

В первом примере атрибут “*Дети*” содержит множество значений, во втором в таблице хранится иерархический список. При работе

с таблицами в такой форме возникают проблемы выборки информации, ее добавления и изменения. Например, для подсчета количества детей у какого-либо человека необходимо выполнить разбор значения, хранящегося в поле "*Дети*". В примере 2 для вывода списка преподавателей, читающих определенный курс (допускается, что один курс могут читать несколько преподавателей) так же необходимо выполнить набор достаточно сложных операций и т. д.

Для перевода описания какой-либо сущности в первую нормальную форму необходимо выполнить требование атомарности всех атрибутов. Например, для перевода вышеприведенных таблиц в первую нормальную форму необходимо их представить в следующем виде:

Пример 3

ФИО	**Дети**
Иванов И. И.	Сергей
Иванов И. И.	Анна
Петров С. В.	Елена
Сидоров Е. А.	Анна
Сидоров Е. А.	Мария
Сидоров Е. А.	Ксения
Козлов Б. Е.	Владимир
Попугаев Д. М.	Виктор
Попугаев Д. М.	Наталья
Попугаев Д. М.	Ольга

Пример 4

Преподаватель	**Курсы**
Валентинов Д. Е.	Высшая математика
Валентинов Д. Е.	Дискретная математика
Валентинов Д. Е.	Математическая логика
Валентинов Д. Е.	Теория множеств
Блюмберг Р. Э.	Информатика
Блюмберг Р. Э.	Прикладное программирование
Блюмберг Р. Э.	Мульти-медия технологии

Вторая нормальная форма (2НФ)

Таблица находится во второй нормальной форме, если она находится в первой нормальной форме и у нее отсутствуют неполные функциональные зависимости непервичных атрибутов от ключей.

Рассмотрим пример.

Имеется таблица СТУДЕНТЫ(**СТУДЕНТ, КУРС**, КОЛ-ВО ЧАСОВ), в которой хранится список студентов, курсы, которые они прослушали и количество часов, читаемых на каждом курсе. Предположим, что студенту не может читаться один и тот же курс более одного раза, тогда атрибуты СТУДЕНТ и КУРС составят первичный ключ (выделены полужирным шрифтом).

В данной таблице имеются следующие функциональные зависимости:

СТУДЕНТ, КУРС → КОЛ-ВО ЧАСОВ и

КУРС → КОЛ-ВО ЧАСОВ.

В данном случае имеется неполная функциональная зависимость атрибута КОЛ-ВО ЧАСОВ от первичного ключа. Это приводит к следующим аномалиям:

Аномалия добавления: если в учебном плане добавляется новый курс, информация о нем не может храниться в базе до тех пор, пока хотя бы один студент не прослушает его.

Аномалия удаления: если учебный курс отменяется, то информацию о курсе придется удалить из базы данных, несмотря на то, что она, возможно, потребуется позже.

Аномалия изменения: при изменении количества часов, отводимых на курс, требуется просмотр всех записей таблицы для того, что бы отразить изменение количества часов для всех студентов.

Вышеописанные аномалии возникают из-за объединения двух семантически разных фактов в одной структуре.

Для избавления от неполной функциональной зависимости необходимо разделить таблицу на две:

СТУДЕНТЫ(**СТУДЕНТ, КУРС**)

КУРСЫ(**КУРС**, КОЛ-ВО ЧАСОВ)

Количество часов, прочитанных какому-либо студенту можно получить путем соединения двух таблиц по атрибуту товар. Изменение часов, отведенных на курс, потребует изменения только одной записи в таблице КУРСЫ.

Третья нормальная форма (3НФ)

Таблица находится в третьей нормальной форме, если она находится во второй нормальной форме и в ней отсутствуют транзитивные зависимости непервичных атрибутов от ключевых.

Транзитивную зависимость можно определить следующим образом: если A → B, B $\not\rightarrow$ A (при этом B не является ключом), и B → C, то получается транзитивная зависимость A → C.

Рассмотрим следующую таблицу: М_УЧЕБЫ(**СТУДЕНТ**, КАФЕДРА, ЗАВ. КАФЕДРОЙ). Так как студент может учиться толь

ко на одной кафедре, то в данном случае поле СТУДЕНТ будет ключевым. В этой таблице имеются следующие зависимости:

СТУДЕНТ → КАФЕДРА

КАФЕДРА → ЗАВ.КАФЕДРОЙ

Аномалии: в данном случае они те же, что и для таблицы, не соответствующей 2НФ, так как вызывающая их причина одна и та же – наличие двух семантически не связанных фактов в одной структуре; если в данный момент на кафедре не учится не один студент, то в базу нельзя ввести информацию о заведующем кафедры (аномалия добавления); если по какой-либо причине на кафедре временно приостановлено обучение, то после выпуска последнего студента информацию о кафедре и ее заведующем нельзя сохранить в базе (аномалия удаления); если поменялся заведующий кафедрой, то для отображения этих изменений в базе данных необходимо выполнить просмотр всей таблицы (аномалия изменения).

Для устранения вышеописанных аномалий таблица разбивается на две:

М_УЧЕБЫ(**СТУДЕНТ**, КАФЕДРА)

КАФЕДРЫ(КАФЕДРА, ЗАВ.КАФЕДРОЙ)

Нормальная форма Бойса-Кодда (НФБК)

Таблица находится в нормальной форме Бойса-Кодда, если она находится в 3НФ и в ней отсутствуют зависимости первичных атрибутов от непервичных.

Рассмотрим пример: у нас имеется таблица М_РАБОТЫ(**ПРЕПОДАВАТЕЛЬ, КАФЕДРА**, ФАКУЛЬТЕТ). Так как по условиям данного университета любому преподавателю можно преподавать на нескольких кафедрах только одного факультета, то поля ПРЕПОДАВАТЕЛЬ и КАФДЕРА будут ключевыми.

В данном случае имеются следующие функциональные зависимости:

ПРЕПОДАВАТЕЛЬ, КАФДЕРА → ФАКУЛЬТЕТ

ФАКУЛЬТЕТ → КАФДЕРА

Это отношение находится в третьей нормальной форме, так как в нем отсутствуют неполные функциональные и транзитивные зависимости. Однако несмотря на это наблюдаются следующие **аномалии** при работе с таким отношением: невозможно хранить сведения о имеющихся кафедрах на факультете до тех пор, пока в базу не будут внесены данные о сотрудниках данных кафедр (аномалия включения). Если с кафедры уволятся все сотрудники, то сведения о самой кафедре также будут удалены из базы (аномалия удаления). Если меняется название факультета или кафедра переходит в подчинение другого

факультета, то для внесения соответствующих изменений потребуется просмотр всей таблицы (аномалия обновления).

Для устранения указанных аномалий необходимо выполнить декомпозицию исходной таблицы на две таблицы, соответствующие НФБК:

М_РАБОТЫ(**ПРЕПОДАВАТЕЛЬ, КАФЕДРА**)
КАФЕДРЫ(**ФАКУЛЬТЕТ**, КАФЕДРА)

Четвертая нормальная форма (4НФ)

Следующие две нормальные формы связаны с наличием многозначных зависимостей в таблицах. Зависимость является многозначной, если каждому значению A соответствует множество (возможно, пустое) значений B в таблице R, другие атрибуты которой никак не связаны с B. Такая зависимость обозначается $A \Rightarrow B$. Полнее о многозначных зависимостях можно прочитать в 228 Озкарахан.

Таблица находится в четвертой нормальной форме, если она находится в НФБК и для любой многозначной зависимости $A \Rightarrow B$ A обязательно содержало бы ключ таблицы.

При несоответствии таблицы 4НФ могут наблюдаться аномалии при обновлении и удалении записей. Например, для рассмотренной выше таблицы:

Для вышеприведенного примера преобразование в 4НФ будет иметь следующий вид:

Пятая нормальная форма (5НФ)

Глава 1.3 Работа с СУБД Interbase 6.x

§ 1.3.1 Работа с базой в режиме файл-сервер и клиент-сервер. Преимущества и недостатки.

В начале развития персональных компьютеров их вычислительная мощность была недостаточна для установки мощных систем управления базами данных. Кроме того, недостаточно были развиты и операционные систем таких компьютеров, что не позволяло с минимальными затратами реализовать мощную СУБД. В связи с этим ведение баз данных на персональных компьютерах выполнялось в режиме файл-сервера, а реализация больших систем хранения и обработки данных осуществлялось на специально разработанных для этой цели компьютерах, стоимость которых была весьма высока.

Однако с течением времени производительность персональных компьютеров постоянно росла, росли и их привлекательность для конечных пользователей, в то время как их цены постоянно снижались. В настоящее время производительность персональных компьютеров сравнима с производительностью так называемых main-frame'ов при их существенно более низкой цене и гораздо более высокой распространенности, что привело к разработке высококачественного как системного (различные варианты операционных систем Windows и Unix), так и прикладного (в том числе и систем управления базами данных) программного обеспечения. В связи с этим давайте рассмотрим основные отличия, достоинства и недостатки архитектур файл-сервер (локальные базы данных) и клиент-сервер.

Основной чертой архитектуры файл-сервер является то, что данные хранятся сами по себе, чаще всего в виде структурированных файлов в файловой системе, поддерживаемой ОС (отсюда, собственно, и название архитектуры), без каких-либо программных средств, обеспечивающих контроль за целостностью хранимых данных. Обработка данных осуществляется специальными прикладными программами, которые обращаются непосредственно к файлам данных через соответствующие сервисы операционной системы. И если даже современные программы (такие, например, как Microsoft Access или Paradox) могут использовать какие-то языки высокого уровня для работы с данными (например, SQL), то работа таких программ все равно происходит в режиме файл-сервера, так как обработка запросов происходит непосредственно в программе клиента, а после выгрузки такой программы не имеется какой-либо возможности контролировать и манипулировать исходными данными.

Отсутствие программы-посредника (как в технологии клиент-сервер) в большинстве случаев существенно снижает требование к аппаратному обеспечению компьютера и приводит к значительному увеличению производительности таких систем обработки данных. Однако при этом следует иметь в виду, что скорость работы программ также сильно зависит и от мастерства разработчиков. К сожалению, вышеописанное утверждение характерно для работы с данными одновременно только одной программы. Требования же большинства реальных задач таковы, что присутствует необходимость одновременной работы с данными нескольких операторов (иногда значительного их числа). Обычно такая задача решается путем создания локальной сети того или иного уровня с выделенным сервером, или используемой в его качестве рабочей станции. Кроме того, современные операционные системы являются многозадачными, то есть позволяют запускать сразу несколько программ, которые могут попытаться одновременно работать с одними и теми же файлами.

Все это привело к разработке механизма блокировок, который в зависимости от используемой операционной системы и локальной базы данных может реализовываться разными способами, однако в любом случае это приводит к некоторому снижению производительности, особенно при операциях обновления индексированных (см. § 1.3.4) таблиц по сравнению с исключительным доступом к данным одного приложения.

Работа в сети в архитектуре файл-сервера требует при операциях выборки данных с условиями отбора пересылать по сети большой объем данных (часто все данные, хранимые в таблицах, используемых в выборке). А так как производительность сети обычно ниже производительности остальных компьютерных систем, то, соответственно, базы данных в режиме файл-сервера быстро приводят к перегрузке сети при незначительном увеличении количества пользователей. Правда, все современные локальные системы управления БД стараются оптимизировать такие запросы и работают по возможности не с самими данными, а с их индексами, которые обычно существенно меньше по объему, что несколько снижает остроту проблемы. Однако при операциях изменения или добавления индексированных данных, файлы, содержащие индексы таких данных, могут блокироваться на достаточно продолжительное время, что снижает производительность системы в целом.

Практический предел размеров сети при использовании системы управления базами данных в режиме файл-сервера определяется производительностью самой сети, размером базы данных и характе

ром обработки данных. Для базы данных в несколько сотен мегабайт, локальной сети 10 Мбит, средней скорости добавления/изменения данных, характерной, например, для небольшого банка и достаточно редким формированием аналитических отчетов можно рекомендовать ограничить максимальное количество активно работающих клиентов числом 20. При увеличении производительности сети нелинейно возрастает количество одновременно работающих клиентов. Например, для рассматриваемого примера при замене сети на 100 Мбит максимальное рекомендуемое количество клиентов составляет 30.

При работе в режиме файл-сервера необходимо давать соответствующие права доступа к файлам данным конечным пользователям системы, что снижает их защищенность от несанкционированного доступа. В принципе, в этой ситуации возможно как изменение данных в самих таблицах с нарушением ссылочной и иной целостности, так и физическое разрушение структуры этих файлов.

В большинстве случаев отсутствуют механизмы разграничения прав пользователей внутри самой базы, что так же снижает ее защищенность.

Часто для обеспечения смысловой целостности базы данных используют так называемые транзакции (см. § 1.3.7). В системах файл-сервер транзакции либо не реализованы, либо их использование связано с определенными проблемами. В частности, если одна из клиентских программ начала транзакцию и ее не завершила в силу внешних причин (например, сбоя электропитания), то часть информации этой незавершенной транзакции будет храниться в базе неопределенное время, вхолостую используя дисковое пространство. Для решения таких проблем необходим либо постоянный административный контроль за базой данных, либо создание специальной программы, которая будет этим заниматься. Все это, в общем, увеличивает издержки, связанные либо с управлением базой данных, либо с ее разработкой.

За редким исключением также отсутствуют такие специализированные утилиты, как утилита создания/восстановления резервных копий, проверки целостности базы данных и т. д., которые существенно упрощают сопровождение уже запущенной в эксплуатацию базы данных.

Основным отличием архитектуры клиент-сервер от вышеописанной файл-серверной архитектуры является обязательное наличие приложения-посредника, который собственно и выполняет различную обработку данных. Это приводит к тому, что клиентские части системы обработки данных становятся более легковесными, что позволяет при разработке гораздо больше внимания уделять интерфейсу пользо

вателя. С другой стороны, наличие промежуточного слоя между клиентом и данными приводит к существенному снижению производительности при однопользовательском режиме работы.

Преимущества архитектуры клиент-сервер начинают проявляться при работе большого количества пользователей с достаточно объемными данными. В этом случае в связи с тем, что по сети передаются только необходимые для клиента данные, которые в соответствии с его запросом были предварительно выбраны на сервере, существенно снижается нагрузка на сеть. Централизация обработки данных также позволяет организовать планирование обращений к базе, высококачественную оптимизацию запросов пользователей, что при больших размерах базы и большом количестве пользователей так же повышает производительность. Обычно с серверами баз данных так же поставляются различные программы администрирования, что существенно облегчает сопровождения баз данных.

Серверы баз данных могут быть организованы по-разному, однако в последнее время наиболее широкое распространение получили SQL-сервера, основанные на выполнении простых выражений, написанных на специальном языке – языке структурированных запросов (Structured Query Language – SQL), что и дало название этой группе серверов.

В общем, преимущества архитектуры клиент-сервер существенно перевешивают присущие ей недостатки, что приводит к бурному развитию серверов баз данных, которые начинают использоваться даже и в однопользовательском режиме, а также для обслуживания потребностей операционных систем.

Выводы:

1. В однопользовательском режиме работы системы файл-сервер имеют преимущество по производительности перед системами клиент-сервер.
2. При работе в сети, особенно с объемными базами данных, использование сервера БД выглядит в настоящее время существенно более предпочтительным.
3. Разработка сложных баз данных обычно происходит проще в системе клиент-сервер в связи с разделением клиентской части и части, ответственной за выполнение работы с данными.

Рекомендации:

При создании небольших неответственных баз данных, которые используются одним или несколькими пользователями одновременно, проще использовать локальные базы данных со средствами быстрыми разработки, такие как MS Access или Paradox+Delphi и т. д.

В иных случаях рекомендуется использование архитектуры клиент-сервер.

§ 1.3.2 Основные свойства SQL сервера Interbase 6.x

Interbase 6.x – новое поколение SQL сервера фирмы Inprise (бывшая Borland). От аналогичных СУБД других производителей отличается более скромными требованиями к аппаратному обеспечению, а также способом распространения – Interbase версии 6.0 распространяется бесплатно для некоммерческого использования. С точки зрения средств визуальной разработки баз данных в настоящее время Interbase уступает продуктам других известных производителей, однако, этот недостаток компенсируется наличием средств разработки сторонних производителей. Функционально же продукт Inprise является полноценным SQL сервером баз данных с достаточно полной реализацией стандарта SQL-92 и мощным и удобным встроенным процедурным языком.

Одной из отличительных черт сервера является поддержка многоверсионности записей таблиц в базе данных. Многоверсионность подразумевает под собой создания собственного набора измененных в рамках транзакции данных при отсутствии их блокировки на чтение во время работы. Это позволяет существенно снизить вероятность блокировок во время одновременного выполнения нескольких транзакций. Возможно так же подтверждение транзакции без ее закрытия, что избавляет сервер от значительных расходов на создание нового контекста транзакции. Для высоконагруженной базы данных, обслуживающих значительное количество пользователей, такая возможность приводить к существенному повышению производительности. В случае подтверждения транзакции без ее закрытия многоверсионность позволяет читающим транзакциям видеть последние подтвержденные изменения сделанные незакрытой транзакцией. Естественно, что Interbase поддерживает и классический вариант выполнения транзакций с блокировкой записей таблиц.

Interbase 6.x доступен на разных платформах: Windows 9x, Windows ME, Windows NT, Windows 2000, а так же на разных вариантах Unix.

Требования к системам на базе Windows.

- *Операционная система*: Windows NT 4.0 с Service Pack 4 или выше, Windows 2000, Windows 95 с Service Pack 1, Windows 98 или Windows ME.
- *Память*: 16 Мб минимум, 64 Мб рекомендуется для выделенного сервера баз данных.

– *Процессор*: 486DX2 66 МГц минимум, рекомендуется Pentium 100MHz или лучше для многопользовательского сервера.

Поддержка SQL

InterBase соответствует базовым требованиям стандарта SQL-92. Он поддерживает декларативную ссылочную целостность с каскадными операциями обновления/удаления, обновляемые представления данных (views) и внешние объединения. Сервер InterBase поставляется с библиотеками, которые поддерживают разработку клиентских приложений со встроенным SQL и DSQL. На всех платформах InterBase клиентское приложение может обращаться к InterBase API – библиотеке функций, которая позволяет отправлять запросы по обработке данных на сервер.

InterBase также поддерживает расширенные возможности SQL, некоторые из которых упреждают расширение SQL3 к стандартному SQL, в том числе хранимые процедуры, триггеры, роли SQL и поддержку сегментированных Blob (большие бинарные объекты).

Многопользовательский доступ к данным

InterBase обеспечивает одновременный доступ к одной базе данных множеству клиентских приложений. Так же возможен одновременный доступ одного приложения к разным базам данных. SQL триггеры могут информировать клиентов о возникновении каких-либо специальных событий, таких как добавление или удаления записей таблиц и т. д. Разработчики могут также создавать так называемые пользовательские функции (user-defined functions – UDFs), которые в дальнейшем будут храниться вместе с базой, что позволит использовать их любому приложению, имеющему доступ к данной БД.

Управление транзакциями

Клиентское приложение может начать множество одновременных транзакций. InterBase обеспечивает полное и точное управление запуском, подтверждением и откаткой транзакций. Выражения и функции, которые управляют запуском транзакции, также определяют ее поведение. Транзакция в рамках сервера InterBase может быть изолирована от изменений, вносимых другими конкурирующими транзакциями. На все время жизни таких транзакций база данных представляется им как неизменная со времени старта транзакции и в базе видимы только те изменения, которые были сделаны только в рамках этой транзакции. То есть записи, удаленные другими транзакциями, продолжают существовать для такой транзакции, новые записи не видны в рамках этой транзакции, а измененные записи остаются в

своем исходном состоянии. Подробности по управлению транзакциями можно найти в части второй данного пособия.

Многоверсионная архитектура

InterBase предоставляет надлежащую поддержку критичных ко времени транзакций при параллельном и непротиворечивом использовании в смешанном – запросы к данным и их обновлении – окружении. InterBase использует многоверсионную архитектуру, что подразумевает создание и сохранение многих версий каждой записи данных. Путем создания новой версии записи, InterBase позволяет всем клиентам иметь доступ к версии записи в любое время, даже если в это время другой пользователь производит изменение этой записи. InterBase также использует механизм транзакций для изоляции набора изменений базы данных от других модификаций.

Оптимистическая блокировка записей

Оптимистическая блокировка подразумевает блокирование записей только в случае их реального изменения клиентом, а не в момент старта транзакции. Такая технология используется InterBase для обеспечения более высокой производительности базы данных при работе клиентов. В InterBase реализована блокировка на уровне записей, для того, что бы ограничить доступ только к тем записям, которые были изменены клиентским приложением, в отличие от страничной блокировки, которая блокирует все близкорасположенные записи в базе данных. Блокировка на уровне записей снижает вероятность конфликтов при изменении данных разным клиентам в одной и той же таблице. Это приводит к существенному увеличение производительности и снижению количества последовательных операций с базой данных. Однако для специальных случаев в InterBase оставлена возможность и для блокировки таблиц полностью на время выполнения транзакции.

Управление базой данных

В поставку InterBase входят системы управления базой данных и сервером, доступные и как Windows-приложения и в виде утилиты программной строки. Вы можете выполнять задачи администрирования базами данных на локальном InterBase или на InterBase сервере с помощью IBConsole – Windows-приложения, запускаемого на клиентском компьютере. Программы управления сервером позволяют:

- управлять системой безопасности сервера;
- создавать резервные копии базы данных и производить восстановление баз данных из резервных копий;
- выполнять операции по поддержке базы данных;

– просматривать статистику базы данных и менеджера блокировок.

Управление безопасностью сервера

InterBase поддерживает список пользователей и их паролей в специальной базе данных регистрации. База данных регистрации позволяет пользователям подключаться к какой-либо базе данных только в том случае, если комбинация имени пользователя и пароля, получаемых с клиентской стороны, совпадает с допустимыми именами пользователя и пароля, хранящимися в базе регистрации. База данных регистрации находится в файле isc4.gdb, хранящимся на компьютере, который является сервером InterBase. У администратора имеется возможность добавлять пользователей и модифицировать их параметры.

Резервирование и восстановление баз данных

Вы можете резервировать и восстанавливать базы данных, используя IBConsole или утилиту командной строки gbak. Процесс резервирования может выполняться параллельно с другими процессами доступа к базе данных, так как резервирование не требует монопольного доступа к базе данных.

Кроме выполнения задач создания резервных копий важной информации резервирование и восстановление может также использоваться для следующих задач:

– удаление устаревших версий записей базы данных;
– изменение размера страницы базы данных;
– преобразования однофайловой базы данных в многофайловую;
– переноса базы данных с одной операционной системы на другую;

Так же имеется возможность резервировать только описание базы данных для того, что бы иметь возможность создавать пустую базу данных.

Поддержка базы данных

Администратор может подготовить базу данных к закрытию и выполнению задач поддержки базы данных как с помощью IBConsole, так и утилит командной строки. Если в базе данных возникли небольшие проблемы, такие как ошибка записи операционной системы, то Вы можете произвести восстановление базы данных без ее отключения.

Некоторые задачи поддержки базы данных:

– очистка базы данных;
– закрытие базы данных для получения монопольного доступа к ней;

– проверка фрагментов таблиц;
– подготовка разрушенной базы данных к резервированию;
– разрешение незавершенных транзакций двухэтапного завершения;
– проверка и восстановление структуры базы данных;

Просмотр статистики

IBConsole позволяет администратору базы данных отслеживать текущий статус базы данных путем просмотра статистики начальной страницы базы данных и анализа таблиц и индексов.

§ 1.3.3 Работа и управление базой данных: IBConsole.

InterBase поставляется с программой управления IBConsole, имеющей интуитивно понятный пользовательский интерфейс, с помощью которой Вы можете выполнить любую задачу, необходимую для конфигурации и поддержки сервера, создания и администрирования баз данных на сервере и для выполнения SQL запросов в диалоговом режиме. Сюда также входят поддержка безопасности InterBase, управление сервером, базами данных и диалоговый SQL InterBase (ISQL). IBConsole является программой для Windows, однако может работать с серверами Interbase, запущенными под любой поддерживаемой операционной системой.

IBConsole позволяет:

- управлять системой безопасности сервера
- осуществлять резервное копирование/восстановление базы данных
- просматривать статистические данные о работе баз данных и серверов
- выполнять поддержку баз данных, в том числе:
  – проверять целостность баз данных
  – производить очистку мусора в базах данных
  – восстанавливать транзакции, находящиеся в неопределенном состоянии

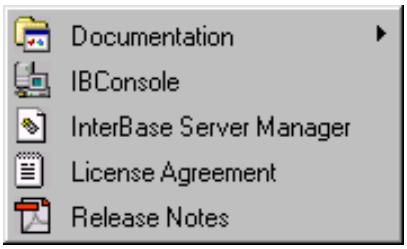

Рис. 1

Для того, чтобы начать работу с IBConsole, выберите пункт IBConsole (рис. 1) из рабочего меню InterBase 6, после чего откроется основное окно IBConsole (рис. 2).

Элементы основного окна Interbase:

– команды меню для выполнения задач администрирования базы данных(DBA) с помощью IBConsole.

– Кнопки на панели инструментов, которые соответствуют основным командам меню. Панели инструментов могут быть как зафиксированными в определенном месте, так и плавающими.
– Иерархическая панель, показывающая иерархию серверов и баз данных, зарегистрированных в IBConsole.
– Рабочая панель, отображающая детальную информацию или позволяющая пользователю выполнять какие-либо действия в зависимости от объекта, выбранного в иерархическом списке.
– Полоса статуса, показывающая текущий сервер и имя входа пользователя, а также подсказку для пунктов меню и панели инструментов.

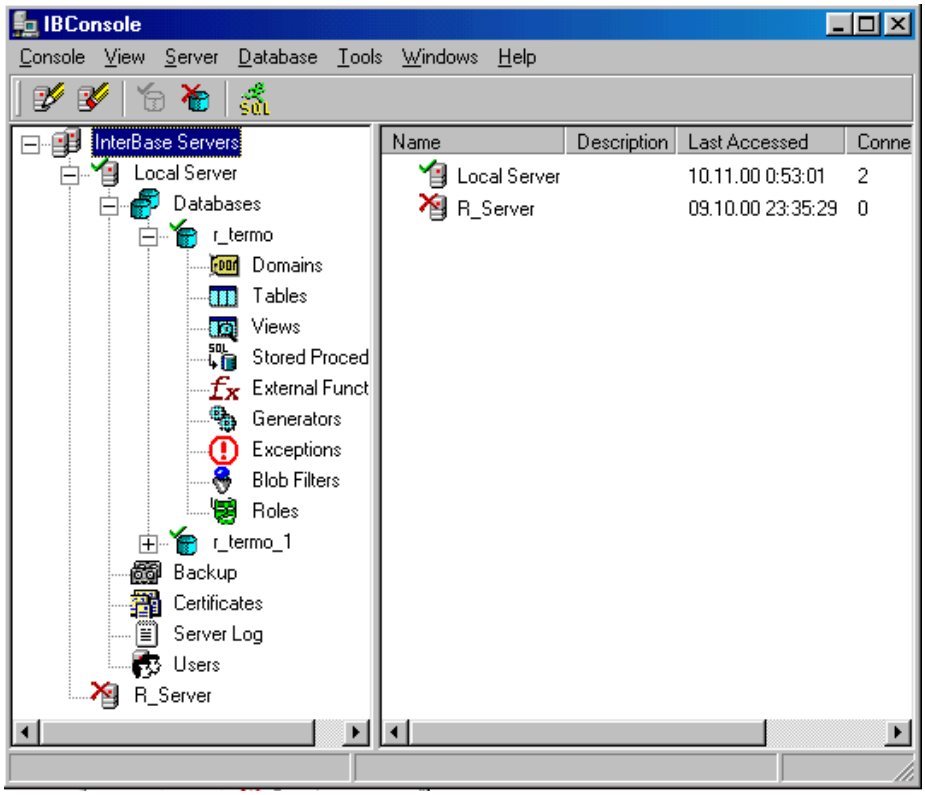

Рис. 2

При нажатии правой кнопки мыши на значке какого-либо объекта иерархического списка открывается всплывающее меню, которое позволяет выполнять определенные действия с выбранным объектом. Ниже приведен список основных действий, которые можно выполнять с объектами базы данных.

InterBase Servers:

– Register – позволяет зарегистрировать новую базу данных (т. е. указать месторасположение и основные характеристики связи с уже существующим сервером).

Уже зарегистрированный сервер:

– Un-Register – позволяет отменить связь программы IBClient с выделенным сервером;
– Login – позволяет подключится к выбранному серверу; при этом в диалоговом окне необходимо ввести имя входа и пароль;
– Diagnose connection – позволяет провести тестирование соединения с данным сервером, либо с определенной базой, расположенной на этом сервере; параметры соединения задаются в диалоговом окне.
– Properties – позволяет просмотреть свойства подключения и, для удаленных серверов, поменять эти свойства; если к серверу было произведено подключение, то так же можно увидеть дополнительную информацию о лицензионном статусе и количестве открытых баз данных.

4IBConsole toolbar

A toolbar is a row of buttons that are shortcuts for menu commands. The following table

describes each toolbar button in detail.

Remove Certificate Remove certificate ID/keys for the current server.

User Security Authorize users on the current server.

View Log file Display the server log for the current server.

Diagnose Connection Display database and network protocol communication

diagnostics.

Server Properties View and update server information for the current server.

Popup command Description

Disconnect Disconnect from the current database.

Maintenance Perform maintenance tasks including: view database statistics,

shutdown, database restart, sweep, and transaction recovery.

Backup/Restore Back up or restore a database to a device or file.

View Metadata View the metadata for the selected database.

Properties View database information, adjust the database sweep interval,

set the SQL dialect and access mode, and enable forced writes.

TABLE 2.2 IBConsole context menu for a connected database icon

Popup command Description

TABLE 2.1 IBConsole context menu for a server icon (continued)

4Tree pane

When you open the IBConsole window, you must register and log in to a local or remote server and then register and connect to the server's databases to display the Tree pane.

See "Connection specification" on page 66 to learn how to register and connect servers and databases.

FIGURE 2.2 IBConsole Toolbar

Button Description

Register server: opens the register server dialog, enabling you to register and login to a local or remote server.

Un-register server: enables you to unregister a local or remote server. This automatically disconnects a database on the server and logout from the server.

Database connect: opens the database connect dialog, enabling you to connect to a database on the current server.

Database disconnect: enables you to disconnect a database on the current server.

Launch SQL: opens the interactive SQL window, which is discussed in detail in Chapter 9: "Interactive Query".

TABLE 2.3 IBConsole standard toolbar

FIGURE 2.3 IBConsole Tree pane

Navigating the server/database hierarchy is achieved by expanding and retracting nodes (or branches) that have subdetails or attributes. This is accomplished by a number of methods, outlined in Table 2.4.

To expand or retract the server/database tree in the Tree pane:

Tasks Commands

Display a server's databases • Left-click the plus (+) to the left of the server icon

• Double-click the server icon

• Press the plus (+) key
• Press the right arrow key
Retract a server's databases • Left-click the minus (–) to the left of the server icon
• Double-click the server icon
• Press the minus (–) key
• Press the left arrow key
TABLE 2.4 Server/database tree commands
Current Ser ver
Current Database
Expand current database to see hierarchy of
tables, views, procedures, functions, and other
database attributes.
Similarly, you can follow these methods to expand or retract any tree branch. Expanding
a database branch displays a list of database attributes. Expanding a table branch displays
a list of table attributes, and so on.
In an expanded tree, click a database name to highlight it. The highlighted database is
the one on which IBConsole operates, referred to as the current database. The current
server is the server on which the current database resides.
The hierarchy displayed in the Tree pane of figure 2.1 is an example of a fully expanded
tree.
g
Expanding the InterBase Server Aliases branch displays a list of registered servers.
g
Expanding a connected server branch displays a list of server attributes, including
Databases, Backups, Users, Certificates, and the Server Log.
g
Clicking on the Database branch displays a list of registered databases on the current
server.
g
Expanding a connected database branch displays a list of database attributes, including

Domains, Tables, Views, Stored Procedures, External Functions, Generators, Exceptions,
Blob Filters, and Roles.
4Work pane
Depending on what item has been selected in the Tree pane, the Work pane gives specific
information or enables you to execute certain tasks.
g
Clicking on the Backup icon displays a list of backup aliases for the current server.
g
Clicking on the Certificates icon displays a list of InterBase certificate keys and IDs for
the current server.
g
Clicking on the Users icon displays a list of users defined on the server.
g
Clicking on a a database attribute icon displays information for that particular attribute.
Clicking on the icon for a database object, such as a table name, in the Work Pane
launches an object viewer specific to that object. These are discussed in “Viewing
metadata” on page 194.
4Standard text display window
The standard text display window is used to monitor database backup and restoration, to
display database statistics and to view server and administration logs.
The standard text display window contains a menu bar, a toolbar with icons for
often-used menu commands, and a scrolling text display area. Figure 7.3, “Database
backup verbose output” on page 151 is an example of the standard text display window.
Elements in a standard text display window:
g
Menu bar A File menu enables you to Save the contents of the window, Print the contents

of the window and Exit from the window. An Edit menu enables you to Copy selected
text in the window to the clipboard, Select All text in the window, and Find a specified
word or phrase within the displayed text.

g

Tool bar Save, Print, and Copy toolbar buttons enable you to save and print the contents
of the text display window as well as copy selected text to the clipboard.

g

Status bar Shows the cursor location, given by line and column, within the text display
window.

InterBase Security

InterBase Security includes server security features that control how a database is
accessed and used.

Server security enables you to:

g

Add a user to the security database

g

Delete a user from the security database

g

Modify user information in the security database

g

Display a list of users in the security database

See Chapter 5, “Database Security,” on page 89 for more details on server security.

Server Management

Server management features enable you to:

g

Register/un-register a server and login/logout a server

g

Manage server/client certificates

g

Retrieve server properties and environment settings

g

Perform server diagnostics

See Chapter 4, “Network Configuration” and Chapter 8, “Database and Server

Statistics” for further information on server management.

Database Management

Database management features offer monitoring and administering of InterBase

databases and servers. These features enable you to:

g

Register/un-register a database and connect/disconnect a database

g

Backup, restore and repair a database

g

View and modify database properties

g

Validate database integrity

g

Perform a database sweep

g

Recover transactions that are “in limbo”

g

Manage the administration log

g

View database statistics

See Chapter 4, “Network Configuration”, Chapter 6, “Database Configuration

and Maintenance” and Chapter 7, “Database Backup and Restore” for more details

on database management.

§ 1.3.4 Создание базы данных на SQL сервере Interbase 6.x

Предварительная подготовка. Перед тем, как создать базу данных, Вы должны определиться со следующими условиями:

1. Место, где будет находиться Ваша база данных. Возможно, этот вопрос стоит обсудить с администратором сети.
2. Желательно определиться с таблицами, которые будут храниться в базе, так как от размера строки (записи) в таблице зависит размер страницы базы данных. Запись, которая целиком не помещается на страницу, требует более чем одного обращения к устройству хранения (которое обычно имеет достаточно низкую производительность), поэтому увеличение размера страницы может ускорить доступ к базе.
3. Насколько большой будет база данных. Производительность работы базы также зависит от количества записей и

размера страницы, так как от размера страницы зависит глубина индекса. Чем меньше глубина индекса, тем более эффективно выполняет операции поиска Interbase.

4. Количество пользователей, которых одновременно будет обслуживать база данных.

Создание базы данных. Базу данных можно создать разными способами: с помощью IBConsole (или средств визуальной разработки других производителей), выполнив команду SQL или выполнив командный файл (скрипт) с помощью IBConsole или isql.

Хотя основные команды работы с базой можно выполнять в диалоговом режиме, рекомендуется делать это с использованием скрипта, так как проще модифицировать существующий файл, чем заново набрать полностью команду SQL.

Поэтому мы начнем с изучения команды SQL. Синтаксис команды выглядит следующим образом:

```
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>][<secondary_file>]
<fileinfo> = LENGTH [=] int [PAGE[S]] | STARTING [AT [PAGE]] int [ <fileinfo>]
```

Рассмотрим вышеприведенные параметры более подробно:

Параметр	Описание
SCHEMA	синоним DATABASE. В команде должно использоваться либо одно, либо другое.
'filespec'	имя файла, где будут храниться данные.
USER 'username'	имя пользователя, с правами которого будет создана база данных. Необязательный параметр; в тех случаях, когда имя пользователя и пароль уже заданы (например, в IBConsole), их можно опустить.
PASSWORD 'password'	пароль пользователя. Необязательный параметр, см. выше.
PAGE_SIZE [=] int	размер страницы базы данных. Указывается целым числом (вместо параметра int), сим

Параметр	Описание
	вол равенства может опускаться. Допустимые значения int 1024 (по умолчанию), 2048, 4096, или 8192.
LENGTH [=] int [PAGE[S]]	длина файла в страницах. Применяется к основному или дополнительным файлам базы данных. В случае создания однофайловой базы данных применяется только к основному файлу. Параметры в квадратных скобках [] можно опускать. ***Внимание!*** Максимальный размер файла для InterBase составляет 4 Гбайта, поэтому произведение размера страницы на количество страниц для каждого файла не должно превышать вышеуказанного числа.
DEFAULT CHARACTER SET charset	установка используемого алфавита по умолчанию для базы данных. Если набор символов не указан, то используется набор по умолчанию NONE. В этом случае сервер не производит какую-либо дополнительную обработку символов и они хранятся в базе как есть. При копировании данных из поля с набором символов NONE в поле с определенным charset или наоборот возникает ошибка транслитерации. Для русского языка рекомендуется в качестве charset использовать WIN1251 (набор символов, используемых в Windows), либо NONE, что оказывает влияние на порядок сортировки. В случае NONE сортировка по некоторым буквам будет выполняться неверно, однако это позволяет использовать русские строковые выражения в хранимых процедурах.
<secondary_file>	необязательный параметр, позволяет создавать многофайловую базу данных. Это необходимо в случае очень больших баз данных, либо из желания увеличить производительность базы данных. В последнем случае базу данных размещают на разных физических дисках, что позволяет прово

Параметр	Описание
	дить загрузку разных данных одновременно (желательно использовать в этом случае SCSI устройства).
<secondary_file> =FILE 'filespec' [<fileinfo>]	после ключевого слова файл указывается расположение дополнительного файла базы данных и информация о нем.
<fileinfo>	параметр задает информацию о дополнительном файле. Здесь указывается либо размер дополнительного файла в страницах (см. описание параметра LENGTH [=] int [PAGE[S]]), либо номер первой страницы, которая будет записываться в этот файл (см. ниже).
STARTING [AT [PAGE]] int	указывается номер начальной страницы, которая будет храниться в дополнительном файле.

В многофайловой БД все файлы за исключением последнего имеют максимальный размер, соответствующий указанному при создании базы данных. Однако их реальный размер может быть меньше указанного в случае если в базе на текущий момент записано мало информации. В частности, сразу после создания базы все файлы имеют минимально необходимый размер для хранения информации. По мере записи информации следующий не заполненный файл увеличивается до указанного в данной команде максимального размера. Последний файл в многофайловой базе данных при заполнении базы увеличивается до максимального размера (в случае однофайловой базы данных увеличивается единственный файл; не имеет смысла в этом случае указывать размер файла).

Примеры создания баз данных:

Однофайловая база данных:

1. create database ‘c:\ib_data\b1.gdb’
   база данных создается в указанном местоположении на локальном сервере.
2. create database ‘server-1:c:\ib_data\b1.gdb’ PAGE_SIZE=8192
   база данных создается на удаленном сервере server-1 с указанным местоположением на нем. Размер страницы базы данных 8 кб. Подключение к серверу проводиться через протокол TCP/IP (протокол определяется по спец. символам

до и после имени сервера: сервер:путь соответствует протоколу TCP/IP, при этом можно указывать как имя компьютера, так и его адрес; \\сервер\путь соответствует протоколу NetBEUI; сервер@путь соответствует протоколу SPX).

3. create database ‘art_srv:c:\ib_data\addresses.gdb’ user ‘sysdba’ password ‘masterkey’ PAGE_SIZE=4096 default character set ‘WIN1251’

   создание базы данных на удаленном сервере art_srv с указанием имени пользователя и пароля (в данном случае sysdba – администратор сервера, masterkey – пароль администратора по умолчанию), размер страницы 4кб, подключение по протоколу TCP/IP, набор символов по умолчанию WIN1251 (русская кодировка Windows).

Многофайловые базы данных:

1. create database ‘server-1:c:\ib_data\b1.gdb’ length 100000
   PAGE_SIZE=8192
   FILE ‘d:\ib_data\b2.gdb’ length 100000
   FILE ‘e:\ib_data\b3.gdb’

   создание базы на удаленном сервере server-1 (протокол TCP/IP), состоящей из 3 файлов, находящихся на разных физических дисках для повышения производительности. Заметьте, для последнего файла не устанавливается параметр длины, так как у последнего файла длина всегда меняется динамически.

2. create database ‘server-1:c:\ib_data\b1.gdb’ PAGE_SIZE=8192
   FILE ‘d:\ib_data\b2.gdb’ starting at 100001 length 100000
   FILE ‘e:\ib_data\b3.gdb’

   пример полностью эквивалентен вышеприведенному, за исключением того, что используется указание начала второго файла вместо спецификации длины основного.

Увеличение размера страницы. Увеличение размера страницы может приводить к повышению производительности по следующим причинам:

- индексы работают быстрее, так как глубина индекса будет минимальной;
- хранение большего количества записей на одной странице более эффективно;

– данные BLOB (Binary Large Object – большой двоичный объект) извлекаются более эффективно с больших по размеру страниц;

Если большинство транзакций работают с небольшим количеством записей, более подходящим может оказаться меньший размер страницы, так меньше данных придется пересылать с диска и обратно и меньше памяти будет требоваться под дисковый кэш.

Изменение размера страниц для существующей базы данных.

1. Сделайте резервную копию базы данных.
2. Восстановите резервную копию с указанием нового размера страницы (параметр PAGE_SIZE).

Более подробно о процедуре создания/восстановления резервной копии см. Operations Guide.

Добавление новых файлов к существующей базе данных. Осуществляется с помощью команды ALTER DATABASE, см. Data Definition Guide и Language References.

База данных с доступом "только чтение". См. Data Definition Guide и Language References.

§ 1.3.5 Типы данных, используемые InterBase

В СУБД InterBase определены следующие типы данных:

Таблица 3

Название	Размер	Диапазон/точность	Описание
smallint	2 байта	–32,768 до 32,767	короткое целочисленное
integer	4 байта	–2,147,483,648 до 2,147,483,647	длинное целочисленное
float	4 байта	1.175×10^{-38} до 3.402×10^{38}	короткое вещественное с однократной точностью (7 цифр) в соответствии со стандартом IEEE
double precision	8 байта	2.225×10^{-308} до 1.797×10^{308}	длинное вещественное с двойной точностью (15 цифр) в соответствии со стандартом IEEE
numeric (precision, scale)	Переменный: 2, 4 или 8 байт	– **длина** – от 1 до 18, определяет точное общее количество знаков, – **точность** – определяет количество знаков после запятой; должно быть меньше или равно длине	– Число с фиксированной точкой – Пример, NUMERIC(10, 3) хранит числа точно в формате ppppppp.sss
decimal (precision, scale)	Переменный: 2, 4 или 8 байт	– **длина** – от 1 до 18, определяет точное общее количество знаков, – **точность** – определяет количество знаков после запятой; должно быть меньше или равно длине	– Число с фиксированной точкой – Пример, DECIMAL(10, 3) хранит числа точно в формате ppppppp.sss

Название	Размер	Диапазон/точность	Описание
date	4 байт	от 1.01.0100 до 29.02.32768	Содержит дату
time	4 байт	от 0:00:00 до 23:59:59	Содержит время
timestamp	8 байт	от 1.01.0100 до 29.02.32768	Содержит и дату и время
char(n)	n символов	от 1 до 32767 байт в зависимости от используемого набора символов объем каждого символа может быть от 1 до 3 байт, общий размер поля не может быть больше 32 Кб.	Текст или строка фиксированной длины
varchar(n)	n символов	от 1 до 32767 байт в зависимости от используемого набора символов объем каждого символа может быть от 1 до 3 байт, общий размер поля не может быть больше 32 Кб.	Текст или строка переменной длины
BLOB	переменный	– отсутствует – Размер сегмента не может превышать 64 Кб	Динамически меняющий размер тип данных для хранения большого объема данных, например, соответствующих графике, оцифрованному звуку, тексту и т. д. Базовым структурным модулем является сегмент Подтип BLOB определяет его содержимое

Ниже приведены основные особенности некоторых типов данных.

Целые типы данных. С целыми типами данных (SMALLINT и INTEGER) можно проводить следующие операции:

- сравнение, используя стандартные операторы (=, <, >, >=, <=); другие операторы, такие как LIKE, CONTAINING и STARTING WITH приводят к предварительному преобразованию типа;
- арифметические операции, такие как сложение, вычитание, умножение и деление;
- преобразования; при выполнении смешанных операций с разными типами данных, InterBase автоматически приводит приведение типов между целыми, плавающими и строковыми типами; при сравнении с численных данных с другими типами сначала проводится преобразование к численному типу, а уже затем соответствующая операция;
- сортировка; по умолчанию запрос возвращает строки в том порядке, как они хранятся в базе данных; с помощью конструкции ORDER BY оператора SELECT строки могут быть отсортированы в возрастающем или убывающем порядке;

Типы данных с фиксированной точкой. Такие типы (NUMERIC и DECIMAL) данных хорошо использовать для хранения значений, требующих фиксированную точность, например, денежные значения. Их различие между собой состоит в том, как интерпретируется длина и точность. NUMERIC всегда выделяет указанную длину под число, а DECIMAL по возможности может хранить число и больше.

Например, поле или переменная типа NUMERIC(5,2) будет хранить числа формата ppp.ss, то есть числа от 0 до 999.99. В то же время DECIMAL(5,2) будет хранить числа формата ppp.ss, но при возможности туда будут записываться и большие числа. Это зависит от того, в каком внутреннем формате хранятся эти типы. При длине до 4 они хранятся как тип SMALLINT, при длине до 9 включительно они хранятся как INTEGER, при большей длине как INT64 (внутренний тип данных, 8-байтное целое). То есть DECIMAL(5,2) будет хранится в базе как INTEGER и потенциально может запоминать числа до 10 цифр длиной, т. е. в такое поле можно вводить числа до 9999999.99 (вернее, до 21474836.47). Однако в реальности в версии InterBase 6.1 оба типа ведут себя одинаково.

Рекомендуется использовать эти типы для хранения денежных значений, так как расчеты с ними требуют ограниченной точности (от 2 до 4 знаков после запятой), при большей точности могут возникать проблемы при работе с бухгалтерскими документами. Так же можно использовать эти типы для хранения больших целых значений, когда необходимы абсолютно точные вычисления, в частности, с их помощью можно организовывать счетчики для очень больших таблиц.

Типы данных с плавающей точкой. InterBase поддерживает два типа с плавающей точкой: FLOAT и DOUBLE PRECISION, которые отличаются только точностью. Их рекомендуется использовать в областях, требующих очень большой диапазон данных, в частности, при научных расчетах. С ними допустимы любые арифметические выражения.

Типы данных даты/времени. Три типа даты и времени (DATE, TIME и TIMESTAMP) позволяют хранить соответственно только дату, только время, или и то и другое вместе. В InterBase также встроены следующие функции для работы с датой/временем:

NOW – возвращает текущую дату и время;

CURRENT_DATE – возвращает текущую дату;

CURRENT_TIME – возвращает текущее время;

CURRENT_TIMESTAMP – возвращает текущую дату и время;

EXTRACT(part FROM value) – возвращает часть даты или времени из соответствующего поля или переменной (см. таблицу 4).

Таблица 4

Значение параметра part	Извлекаемая часть	Тип	Представление
YEAR	год	smallint	от 0 до 5400
MONTH	месяц	smallint	от 1 до 12
DAY	день	smallint	от 1 до 31
HOUR	час	smallint	от 0 до 23
MINUTE	минута	smallint	от 0 до 59
SECOND	секунда	decimal(6,4)	от 0 до 59.9999
WEEKDAY	день недели	smallint	от 0 до 6 (воскресенье, понедельник и т. д.
YEARDAY	день года	smallint	от 1 до 365

Можно выполнять явное преобразование типов данных даты/времени в текстовый формат или в другой формат даты/времени. Преобразование в целочисленный формат или формат с плавающей точкой невозможно. Пример преобразования:

SELECT CAST (timestamp_col AS DATE) AS CHAR(10) FROM table1;

При этом длина строковой переменной или поля должна быть не меньше 24 символов для типа TIMESTAMP, для остальных типов она может быть меньше. В результате получается строка следующего вида

Таблица 5

Приведение от	Требуемая длина строки	Результирующий формат
TIMESTAMP	>=24	YYYY-MM-DD HH:MM:SS.тысячные
DATE	>=10	YYYY-MM-DD
TIME	>=13	HH:MM:SS.тысячные

В случае, если длина строки не достаточна для заполнения возникает ошибка усечения и операция не выполняется.

Преобразование из одного типа даты/времени в другой происходит правилам, приведенным в таблице 6.

При добавлении данных с помощью команды SQL INSERT, содержащих дату, допустимы следующие строковые форматы[1]:

[1] dd – день месяца
mm – месяц
yy – год, две цифры

'yyyy-mm-dd'	'yyyy/mm/dd'	'yyyy mm dd'	
'yyyy:mm:dd'	'yyyy.mm.dd'		
'mm-dd-yy'	'mm-dd-yyyy'	'mm/dd/yy'	'mm/dd/yyyy'
'mm dd yy'	'mm dd yyyy'	'mm:dd:yy'	'mm:dd:yyyy'
'dd.mm.yy'	'dd.mm.yyyy'		
'dd-xxx-yy'	'dd-xxx-yyyy'	'xxx-dd-yy'	'xxx-dd-yyyy'
'dd xxx yy'	'dd xxx yyyy'	'xxx dd yy'	'xxx dd yyyy'
'dd:xxx:yy'	'dd:xxx:yyyy'	'xxx:dd:yy'	'xxx:dd:yyyy'

Таблица 6

Преобразование из	**Преобразование в**		
	TIMESTAMP	**DATE**	**TIME**
TIMESTAMP	Полное преобразование	Копируется только дата	Копируется только время
DATE	Копируется дата, время устанавливается в 0:00:00.0000	Полное преобразование	Ошибка
TIME	Копируется время, дата устанавливается в текущую на сервере	Ошибка	Полное преобразование

Некоторые примеры добавления в таблицу записи с датой 22.01.1943:

```
INSERT INTO t1 VALUES ('1943-01-22');
INSERT INTO t1 VALUES ('01/22/1943');
INSERT INTO t1 VALUES ('22.01.1943');
INSERT INTO t1 VALUES ('jan 22 1943');
```

Следующий пример введет дату 22.01.2043:

```
INSERT INTO t1 VALUES ('01/22/43');
```

Возможные сочетания операций с использованием типов даты/времени приведены в таблице 7.

Таблица 7

Операнд 1	**Оператор**	**Операнд 2**	**Результат**
DATE	+	DATE	Ошибка
DATE	+	TIME	TIMESTAMP (соединение даты и времени)
DATE	+	TIMESTAMP	Ошибка
DATE	+	Число	Дата + количество дней, дробная часть числа иг

yyyy – год, четыре цифры

xxx – месяц, три первые буквы английского названия

Операнд 1	Оператор	Операнд 2	Результат
			норируется
TIME	+	DATE	TIMESTAMP (соединение даты и времени)
TIME	+	TIME	Ошибка
TIME	+	TIMESTAMP	Ошибка
TIME	+	Число	Время + количество секунд
TIMESTAMP	+	DATE	Ошибка
TIMESTAMP	+	TIME	Ошибка
TIMESTAMP	+	TIMESTAMP	Ошибка
TIMESTAMP	+	Число	Дата + число (целая часть – число дней, дробная – часть от суток)
DATE	–	DATE	Разница в днях между датами (DECIMAL(9,0))
DATE	–	TIME	Ошибка
DATE	–	TIMESTAMP	Ошибка
DATE	–	Число	Дата – количество дней, дробная часть игнорируется
TIME	–	DATE	Ошибка
TIME	–	TIME	Разница по времени в секундах (DECIMAL(9,4))
TIME	–	TIMESTAMP	Ошибка
TIME	–	Число	Время – количество секунд
TIMESTAMP	–	DATE	Ошибка
TIMESTAMP	–	TIME	Ошибка
TIMESTAMP	–	TIMESTAMP	Разница между двумя датами (DECIMAL(18,9)) – количество дней и доля суток
TIMESTAMP	–	Число	Дата – число (целая часть – число дней, дробная – часть от суток)

Использование агрегирующих функций с полями типа даты/времени. Допускается использование следующие агрегирующие функции с полями типа даты/времени: MIN(), MAX(), COUNT(), в том числе с параметром DISTINCT, а так же операцию группировки для таких полей. Попытки использования функций SUM() или AVG() к полям типа даты/времени приводят к возникновению ошибки.

Текстовые типы данных. В Interbase имеется несколько типов текстовых данных. Подробнее остановимся на двух из них: CHAR(n) и VARCHAR(n). По идее первый тип – это строка фиксированной длины, а второй – соответственно динамической. Поэтому при записи в поле с типом CHAR(n) строки короче n пустое место должно быть заполнено заполнителем (обычно пробел). При записи же в поле с типом VARCHAR(n) в базе хранится актуальное количество символов (в соответствии с длиной введенной строки). Реально же InterBase оптимизирует использование дискового пространства и хранит CHAR так же как и VARCHAR, однако при извлечении данных к полю типа CHAR добавляются пробелы в конец строки. Я рекомендую всегда использовать тип VARCHAR.

Для текстовых типов данных можно задать используемый набор символов. Если набор символов не задан, то используется набор символов по умолчанию, который задается при создании базы данных (см. § 1.3.4). Заданный набор символов влияет на порядок сортировки и сравнения текстовых типов данных. Кроме того, при копировании текста из переменной/поля с одним набор символов в переменную/поле другого набора символов Interbase проводит транслитерацию (корректное преобразование из одного набора символов в другой). Отдельно стоит остановиться на наборе символов NONE. Такой набор символов не подразумевает специальных мер при сортировки и сравнении текстовых строк, поэтому в некоторых случаях эти операции могут выполняться неправильно. В частности, для русского языка в кодировки WIN1251 будет неправильно выполняться сортировка и сравнение строк, содержащих буквы ‘Ё’ или ‘ё’. Это происходит из-за того, что коды этих символов в WIN1251 расположены не упорядоченно по алфавиту; кроме того, при выполнении этих операций для символов в разных регистрах будут выполняться следующие условия: Ё < ё < А < Б < В < … < Я < а < б < в < … < я. Кроме того, при копировании данных из поля с набором символов NONE в поле с любым другим набором символов возникает ошибка транслитерации

Однако использование набора символов NONE позволяет использовать в хранимых процедурах текстовые константы на русском языке, что иногда является определяющим. Это связано с тем, что

системные таблицы имеют по умолчанию набор символов NONE и при любом считывании хранимых процедур из таких таблиц производится попытка транслитерации к набору символов по умолчанию, используемых в базе данных; если в теле хранимой процедуры используются символы, выходящие за первые 128 символов из таблицы кодировки (то есть имеющие символы других языков, кроме английского) и база данных имеет набор символов по умолчанию, отличный от NONE, то возникает ошибка транслитерации и такая хранимая процедура не выполняется.

Размер, который будет занимать в базе текстовое поле, зависит от используемого набора символов. Для большинства кодировок достаточно 1 байта на символ, однако некоторые кодировки, в частности, кодировка Unicode, требуют больше байт на символ (в текущей реализации это 3 байта/символ). Соответственно, максимальная длина полей в символах зависит от набора символов, так как максимальный размер текстового поля ограничен 32767 байт. Поэтому для однобайтовых набор символов максимальная длина текстового поля будет 32767, а для трехбайтовых (например, UNICODE_FSS) будет 10922 (32767 деленное на цело на 3). Список допустимых наборов символов и занимаемого ими места можно найти в *Data Definition Guide*.

Типы данных BLOB. Такие типы поддерживают очень большие объекты, хранящиеся в базе данных, которые необходимы для хранения растровых и векторных изображений, звуковых файлов, видеофрагментов и книг и другой мультимедиа информации. Так как BLOB может хранить существенно разную информацию, то для доступа к таким полям используются специальные методы, во всем остальном же они эквивалентны обычным полям базы данных. Используйте такие поля для того, что бы избежать ссылок на файлы, не входящие в базу данных.

BLOB поля описываются в базе точно так же, как и обычные:

```
CREATE TABLE PROJECT
    (PROJ_ID PROJNO NOT NULL,
    PROJ_NAME VARCHAR(20) NOT NULL UNIQUE,
    PROJ_DESC BLOB,
    TEAM_LEADER EMPNO,
    PRODUCT PRODTYPE,
    ...);
```

Однако данные для таких полей хранятся не в самой таблице, а отдельно в файле(ах) базы данных в сегментах, объединенных в цепочки, а в самой таблице хранится только ссылка на эти данные. При обращении к данным BLOB InterBase читает/пишет один сегмент за

операцию, что повышает производительность. При описании поля типа BLOB может быть задан размер сегмента по умолчанию, однако реальный размер сегмента может переопределяться в процессе записи:

```
CREATE TABLE TABLE2
    (BLOB1 BLOB,
    BLOB2 BLOB SEGMENT SIZE 512);
```

При создании поля BLOB можно также задать подтип, который определяет категорию хранимых в поле данных. В InterBase заранее предописаны следующие подтипы:

Таблица 7

Подтип	Описание
0	Бесструктурный, в основном используется для хранения двоичных данных или данных неопределенного типа
1	Текст
2	Двоичное языковое представление (BLR)
3	Список прав доступа
4	Зарезервировано для будущего использования
5	Закодированное описание метаданных текущей таблицы
6	Описание распределенных транзакций, которые закончились нерегулярно

При создании поля BLOB можно описать свой подтип. Никаких специальных средств по контролю за правильным использованием подтипа данных в InterBase нет, поэтому контроль за соответствие содержимого поля BLOB несет само приложение.

§ 1.3.6 Создание таблиц с использованием SQL

Для создания таблиц на SQL сервере всегда можно воспользоваться командами языка SQL, кроме того, часто либо производители SQL сервера, либо сторонние поставщики поставляют средства визуального построения таблиц, запросов и т. д. К сожалению, в поставки сервера Interbase 6.x нет средств визуального построения базы данных (IBConsole позволяет просматривать базу данных и изменять сами данные, а средства редактирования структуры базы отсутствуют), поэтому мы сначала рассмотрим создание таблиц с помощью SQL, а затем, в конце данной главы, средство визуальной работы с базами данных Interbase QuickDesk фирмы Electronic Microsystems.

Базовый синтаксис SQL для создания таблиц выглядит следующим образом (полный вариант смотрите в документации по Interbase):

```
CREATE TABLE имя_таблицы (<описание_поля> [, < описание_поля > |<огр_таблицы> …]);
```

< описание_поля > = имя_поля {<тип_данных> |
COMPUTED [BY] (<выраж-е>) | домен} [DEFAULT { literal |
NULL | USER}] [NOT NULL] [<огр_поля>] [COLLATE поря-
док_сортировки]

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

<разм_массива> = [[x:]y [, [x:]y …]]

<выраж-е> = Допустимое SQL выражение, возвращаю-
щее одиночное значение.

< огр_поля >= [CONSTRAINT constraint]
{UNIQUE|PRIMARY KEY|REFERENCES др_таблица
[(др_поле)][ON DELETE {NO ACTION|CASCADE|SET
DEFAULT|
SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET
NULL}]
|CHECK (<условие>)}

<огр_таблицы>= [CONSTRAINT constraint] {{PRIMARY
KEY|UNIQUE} (поле [, поле …])
|FOREIGN KEY(поле [, поле …]) REFERENCES
др_таблица(др_поле, др_поле, …)
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET
NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET
NULL}]
| CHECK (<условие>)}

<условие> = <знач> <operator> { < знач > | (<select_one>)}
|< знач > [NOT] BETWEEN < знач > AND < знач >
|<знач> [NOT] LIKE <знач> [ESCAPE <знач>]
|<знач> [NOT] IN (<знач> [, <знач> …] | <select_list>)

|<знач> IS [NOT] NULL
| <знач> {[NOT]{ = | < | > | >= | <=} {ALL |SOME
|ANY}(<select_list>)
| [NOT] EXISTS (<select_expr>)
| [NOT] SINGULAR (<select_expr>)
|<знач> [NOT] CONTAINING <знач>
|<знач> [NOT] STARTING [WITH] <знач>
|NOT <условие>
|<условие> OR < условие >
|< условие > AND < условие >

<знач> = { поле [<разм_массива>] |:переменная | <константа> | <выраж-е> | <функция> | UDF ([<знач> [, <знач> …]]) | NULL | USER | RDB$DB_KEY | ? }

[COLLATE порядок_сортировки]

<константа> = число | 'строка' | набор_символов 'имя_набора'

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

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

<select_one> = SELECT по одному столбцу; должно возвращаться ровно одно значение.

<select_list> = SELECT по одному столбцу; возвращается любое количество значений.

<select_expr> = SELECT по нескольким столбцам; возвращается любое количество значений.

Разберем вышеприведенный синтаксис более подробно. Основой построения таблицы является команда CREATE TABLE, в качестве параметров которой указывается имя таблицы (которое должно быть уникальным в пределах базы данных среди наименований других таблиц, хранимых процедур и отображений), а так же список полей создаваемой таблицы.

Поля. При описании поля вводится его имя, которое должно быть уникальным в пределах таблицы и определяется тип данных, значение в поле по умолчанию, ограничение на значение и порядок сортировки.

Тип данных определяется либо его непосредственным указанием (см § 1.3.5), либо возможностью вычислить каким-либо образом значение поля, либо доменом, который описывает часто используемые типы данных.

Пример 1:

```
CREATE TABLE simple(
    id INTEGER,
    name VARCHAR(50));
```

В случае вычисляемого поля его тип будет определен автоматически исходя из наибольшей мощности типов, использующихся в выражении. При этом допустимы любые арифметические операций, совместимые с типами данных, используемых в выражении. Если в выражении используются обращения к другим полям этой таблицы, то они должны быть описаны до вычисляемого поля. Так же в выражении не могут быть использованы поля типа BLOB и оно должно возвращать одиночное значение, а не массив.

Пример 2:

```
CREATE TABLE simple1(
    id INTEGER,
    sum1 NUMERIC(10,2),
    sum2 NUMERIC(15,3),
    total COMPUTED (sum1+sum2));
```

Для задания значения по умолчанию для поля необходимо использовать конструкцию DEFAULT, в которой можно указать либо константу, совместимую с объявленным для поля типом данных, либо описателем NULL задать, что по умолчанию поле имеет пустое значение, либо с помощью описателя USER определить, что по умолчание в данное поле будет заносится имя пользователя, производящего вставку новой записи. В последнем случае поле должно иметь соответствующий (то есть текстовый) тип данных.

Пример 3:

```
CREATE TABLE simple2(
    id INTEGER,
    sum1 NUMERIC(10,2) DEFAULT 0,
    sum2 NUMERIC(15,3) DEFAULT NULL,
```

u_name varchar(20) **DEFAULT USER**);

При отсутствии явного указания поля в Interbase могут иметь пустые значения, то есть пользователь может не вносить в них информацию. Однако достаточно часто имеются поля, в которых информация обязательно должна присутствовать. Для этого используется ключевое слово NOT NULL.

Пример 4:

```
CREATE TABLE simple3(
    id INTEGER NOT NULL,
    sum1 NUMERIC(10,2) DEFAULT 0,
    sum2 NUMERIC(15,3) DEFAULT NULL,
    u_name varchar(20) DEFAULT USER);
```

Ограничения целостности. Часто возникает необходимость задавать также различные ограничения для значения, которое будет хранится в поле. Это могут быть ограничения на уникальность и межтабличную целостность (см. далее), а так же логические выражения, непосредственно накладывающие ограничения на значение. Последние вводят ключевым словом CHECK, после которого в круглых скобках идет логическое выражение, ограничивающее возможное значение.

Пример 4:

```
CREATE TABLE simple3(
    id INTEGER NOT NULL,
    sum1 NUMERIC(10,2) DEFAULT 0 CHECK(sum1>=0),
    sum2 NUMERIC(15,3) DEFAULT NULL CHECK((sum2 IS
            NULL) OR (sum2 BETWEEN 0 AND sum1)),
    u_name varchar(20) DEFAULT USER);
```

При этом следует обратить внимание на то, что для таблицы допустимо указывать ограничения как в описании поля, так и в описании таблицы в целом. В последнем случае описание ограничения следует после описания всех полей. С точки зрения реализации базы данных эти ограничения вполне одинаковы, однако с точки зрения логики структуры базы данных ограничения на значения, связанные только с одним полем таблицы необходимо описывать в конструкции *огр_поля*, а связанные одновременно с несколькими полями таблицы – в конструкции *огр_таблицы*.

При создании различных ограничений целостности допустимо не указывать имя ограничений, при этом сервер автоматически создаст уникальный идентификатор для такого ограничения. В общем

случае однако гораздо удобнее задавать имя для различных ограничений, что позволит проще понимать сообщения об ошибках при отладке базы данных и облегчить при необходимости модификацию структуры базы. Для задачи имени ограничений PRIMARY KEY, UNIQUE, FOREIGN KEY (REFERENCES для ограничений в описании поля) и CHECK следует использовать предложение CONSTRAINT имя. Рассмотрим пример описания различных ограничений с заданием для них имен:

```
CREATE TABLE simple3(
  id INTEGER NOT NULL CONSTRAINT c1 PRIMARY KEY,
  sum1 NUMERIC(10,2) DEFAULT 0 CONSTRAINT c2
       CHECK(sum1>=0),
  sum2 NUMERIC(15,3) DEFAULT NULL,
  u_name varchar(20) DEFAULT USER,
  CONSTRAINT c3 CHECK((sum2 IS NULL) OR (sum2
       BETWEEN 0 AND
  sum1)),
  CONSTRAINT c4 FOREIGN KEY (u_name) REFERENCES
  other_table(name));
```

В этом примере созданы четыре ограничения с именами *c1*, …, *c4*, при этом ограничение *c1* является ограничением уникальности (первичный ключ), *c2* и *c3* – проверочные ограничения на уровне поля и таблицы соответственно, а *c4* – ограничение ссылочной целостности.

В Interbase используется достаточно богатый набор проверочных выражений. Это и обычные операторы условий и более мощные конструкции, позволяющие также использовать в условиях и запросы SQL для данных, хранящихся в других (а так же и в этой же таблице). Эти же операторы также используются в запросах на выборку и модификацию данных, поэтому рассмотрим их более подробно:

1. *знач* [NOT] BETWEEN <*знач1*> AND <*знач2*> – *знач*, которое может являться в принципе любым допустимым выражением, возвращающим значение сравнимого типа, должно находится в промежутке от *знач1* до *знач2*, включая их. В случае наличия ключевого слова NOT проверяется **не** вхождение *знач* в указанный диапазон.

   Например, мы хотим проверить, входит ли значение поля в диапазон от 10000 до 35000:

   ```
   salary BETWEEN 10000 AND 35000.
   ```

   Или наоборот, мы хотим убедится что значение не входит в этот диапазон:

salary NOT BETWEEN 10000 AND 35000.

2. *знач* [NOT] LIKE < *знач1*> [ESCAPE < *знач2*>] – используется для поиска по шаблону. Если одно из значений в этом выражении не текстового типа, то оно приводится к нему. В качестве групповых символов используются следующие знаки “%” – любое количество (в том числе и ноль) любых символов и “_” – любой одиночный символ в указанной позиции. Например, для поиска всех людей, фамилия которых начинается на ‘Ва’ можно использовать оператор

   fio LIKE ‘Ва%’

   Если при поиске нужно указать один из групповых символов, то следует воспользоваться часть ESCAPE данного выражения. Например, если мы хотим искать строку, содержащую два символа процента подряд:

   – в шаблоне перед символом процента, который используется для поиска, необходимо указать какой-либо другой символ, например, ‘@’;
   – после ключевого слова ESCAPE следует указать этот символ (в данном случае ‘@’); это говорит о том, что следующий за ним символ нужно использовать как есть;

   Для данного примера оператор LIKE будет выглядеть следующим образом:

   s_data LIKE ‘%@%@%%’ ESCAPE ‘@’;

   То есть каждый символ ‘%’ после символа ‘@’ будет трактоваться не как групповой, а как обычный.
3. *знач* [NOT] IN (<*знач1*> [, <*знач2*> …] | <*select_list*>) – *знач* должно находится в явно указанном наборе значений, или в наборе значений, возвращаемых оператором SELECT. Примеры:

   id IN (1, 5, 25, 13);

   id_u IN (SELECT id FROM users WHERE name LIKE ‘Бо%’);

   При использовании оператора NOT проверяется, что *знач* **не** входит в указанный тем или иным способом набор.
4. <*знач*> IS [NOT] NULL – *знач* должно быть пустым, или наоборот, содержать какие-либо данные.
5. <*знач*> {[NOT]{ = | < | > | >= | <=}{ALL |SOME |ANY} (<*select_list*>) – проверяет, выполняется ли указанное условие сравнения для *знач* и всех (в случае ALL) или нескольких (в случае SOME или ANY) значений, возвращаемых

оператором SELECT. При наличии ключевого слова NOT проверяется **не** выполнение указанного условия. Примеры:

- sum > ALL (select sum from s_list where type=’рабочие’ – условие выполнится только в том случае, если значение поля sum будет больше всех значений, которые вернул подзапрос (в данном случае проверяется, что зарплата превышает зарплату любого рабочего)
- sum >= SOME (select sum from s_list where type=’начальство’ – условие выполнится только в том случае, если значение поля sum будет больше либо равно хотя бы одного значения из тех, которые вернул подзапрос (в данном случае проверяется, что зарплата равна или выше зарплаты какого-либо начальника)

6. [NOT] EXISTS (<*select_expr*>) – данный оператор проверяет, возвращает ли указанный в качестве параметра запрос какие-либо записи. С ключевым словом NOT проверяется, что указанный запрос не возвращает ни одной записи. Конструкция SELECT, указанная в подзапросе, обязательно должна возвращать все поля, то есть использовать символ ‘*’. Данная конструкция, как и следующая, чаще используется не при создании таблиц в условиях на значения полей, а при выборке и модификации данных (см. ниже) в условиях отбора. Указанный ниже пример выбирает все страны, на территории которых есть реки:

```
SELECT country
FROM countries c
WHERE EXISTS (SELECT * FROM rivers r
    WHERE r.country = c.country);
```

7. [NOT] SINGULAR (<*select_expr*>) – данная конструкция проверяет, возвращает ли указанный подзапрос ровно одну строку. Так же как и для предыдущего оператора, конструкция SELECT в подзапросе должна возвращать все поля. В качестве примера приведем запрос, который выбирает страны с одной единственной рекой:

```
SELECT country
FROM countries c
WHERE SINGULAR (SELECT * FROM rivers r
WHERE r.country = c.country);
```

8. <*знач*> [NOT] CONTAINING <*знач1*> – эта конструкция проверяет, что в выражении *знач* содержится подстрока *знач1*, при этом регистр для данного оператора не имеет значения, т. е. строки ‘Петров’, ‘петров’ и ‘ПЕТРОВ’ равнозначны (при условии, что используется соответствующий набор символов, для русского языка это WIN1251). Может также использоваться с полями типа BLOB для посегментного поиска указанной подстроки. Следующий пример возвращает истину, если поле name содержит подстроку ‘ова’:
   name CONTAINING ‘ова’.
9. <*знач*> [NOT] STARTING [WITH] <*знач1*> –этот оператор производит проверку, начинается ли *знач* со строки *знач1*. Нижеприведенный пример проверяет, начинается ли содержимое поля name со строкового значения ‘Пет’:
   name STARTING WITH ‘Пет’.

Изменение таблицы. Для изменения структуры уже созданной таблицы используется команда ALTER TABLE. Она позволяет выполнять следующие действия:

- добавлять новые поля в таблицу;
- удалять поле из таблицы;
- удалять ограничения целостности таблицы или колонки;
- изменять имя поля, его тип и позицию.

Можно выполнить любое количество вышеприведенных операций в одной команде. Таблица может быть изменена ее создателем, администратором базы данных и, для некоторых систем, любым пользователем, имеющим привилегии root.

Изменение поля таблицы. Синтаксис команды ALTER TABLE для изменения поля выглядит следующим образом:

ALTER TABLE table ALTER [COLUMN] <имя_поля> <модификатор>
<модификатор> = <новое_имя> | <новый_тип> |
<новое_расположение>
<новое_имя> = TO <имя_поля>
<новый_тип> = TYPE <тип> | <домен>
<новое_расположение> = POSITION целое

Здесь предложение *новое_имя* соответствует изменению названия поля, *новый_тип* – изменению типа поля и *новое_расположение* – изменению места, на котором находится поля в таблице. Подробные примеры приведены ниже.

Перед тем, как произвести изменение поля, необходимо выполнить следующие пункты:

1. Убедиться, что у Вас есть необходимые права доступа.
2. Сохранить существующие данные.
3. Удалить ограничение, наложенные на поле.

Перед изменение поля необходимо сохранить содержащиеся в нем данные, так как в противном случае они могут быть утеряны. Сохранение данных поля и последующая затем модификация требуют выполнения следующей последовательности действий:

1. Добавить временное поле в таблицу, описание которого совпадает с описанием поля, которое мы собираемся изменить.
2. Скопировать данные из текущего поля во временное;
3. Изменить поле.
4. Скопировать данные назад из временного поля в модифицированное.
5. Удалить временное поле.

Пример:

1. Создаем временное поле:
   ALTER TABLE EMPLOYEE ADD TEMP_NO CHAR(3);
2. Копируем данные во временное поле:
   UPDATE EMPLOYEE
   SET TEMP_NO = OFFICE_NO;
3. Изменяем основное поле:
   ALTER TABLE EMPLOYEE ALTER OFFICE _NO TYPE CHAR(4);
4. Копируем данные из временного поля:
   UPDATE EMPLOYEE
   SET OFFICE_NO = TEMP_NO;
5. Удаляем временное поле:
   ALTER TABLE EMPLOYEE DROP TEMP_NO;

В случае, если операция модификации завершилась неудачно, необходимо воссоздать поле и скопировать в него данные из временного поля. Обычно модификация поля проходит удачно, так что создание временного поля является избыточным и требуется только в особенно ответственных случаях.

Примеры модификации поля таблицы:

1. Изменение позиции поля в таблице (перемещаем поле EMP_NO с третьей на вторую позицию):
   ALTER TABLE EMPLOYEE ALTER EMP_NO POSITION 2;
2. Изменение имя поля (меняем название поля EMP_NO на EMP_NUM)

```
ALTER TABLE EMPLOYEE ALTER EMP_NO TO
EMP_NUM;
```

3. Изменяем тип поля (INTEGER на VARCHAR(20)):
```
ALTER TABLE EMPLOYEE ALTER EMP_NUM TYPE
CHAR(20);
```

Допускает преобразование нетекстовых типов в текстовые со следующими ограничениями:

- не преобразуются массивы и поля типа BLOB;
- размер поля не может быть уменьшен;
- новое тип поля должен быть способен хранить существующие данные (то есть быть подходящим по размеру и допускать соответствующее преобразование), в противном случае возникнет ошибка;

Преобразование текстового типа в нетекстовый не допускается.

Внимание! Изменение типа поля может приводить к перестройки индекса, что занимает дополнительно время.

Добавление нового поля в таблицу. Для добавление нового поля используется следующий синтаксис:

ALTER TABLE table ADD <описание_поля>

Определение <*описание_поля*> см. выше в этом параграфе.

Примеры:

1. Добавление одного поля:
```
ALTER TABLE employee ADD emp_no INTEGER NOT
NULL;
```
2. Добавление нескольких полей:
```
ALTER TABLE employee
  ADD emp_no INTEGER NOT NULL,
  ADD full_name COMPUTED BY
  (last_name || ', ' || first_name);
```
3. Добавление полей с указанием ограничений целостности:
```
ALTER TABLE country
  ADD capital VARCHAR(25) UNIQUE,
  ADD largest_city VARCHAR(25) NOT NULL;
```

Добавление новых ограничений к таблице. Команда имеет следующий синтаксис:

ALTER TABLE name ADD [CONSTRAINT constraint] <огр_таблицы>;

где <огр_таблицы> являются ограничениями PRIMARY KEY (первичный ключ), FOREIGN KEY (внешний ключ), UNIQUE (уникальный ключ), или CHECK (проверка), см. выше в этом параграфе; ключевое слово CONSTRAINT служит для указания имени ограничения.

Пример:

```
ALTER TABLE employee
ADD CONSTRAINT dept_no UNIQUE(phone_ext);
```

При добавлении новых ограничений часто бывает удобно задавать имя для этих ограничений, что позволяет очень легко удалять такие ограничения при необходимости. В случае, если имя ограничения не указано, то сервер генерирует имя автоматически и при необходимости удалить ограничение, придется сначала определить его имя, просматривая либо соответствующие системные таблицы, либо свойства таблицы через IBConsole.

Удаление существующего поля. Команду ALTER TABLE можно использовать для удаления полей в таблице. Удалить поле может только владелец таблицы. Если в момент попытки удаления таблицы к ней имеет доступ другой пользователь, то удаление таблицы откладывается до завершения пользовательской транзакции.

Синтаксис команды удаления поля выглядит следующим образом:

```
ALTER TABLE <имя_таблицы> DROP <имя_поля> [,<имя_поля> ...];
```

Примеры:

1. Удаление одного поля:
   ```
   ALTER TABLE employee DROP emp_no;
   ```
2. Удаление двух полей:
   ```
   ALTER TABLE employee
     DROP emp_no,
     DROP full_name;
   ```

Внимание! Нельзя удалить поле, которое используется как часть первичного, уникального или внешнего ключа или входит в ограничение CHECK.

Для удаления поля, которое входит в вышеуказанные ограничения целостности, необходимо сначала удалить эти ограничения.

Удаление ограничений таблицы должно производится в определенном порядке. Например, если Вы хотите удалить первичный ключ у поля, на которое наложено ограничение ссылочной целостности из другой таблицы (так называемый внешний ключ), то перед

этим Вы должны будете удалить соответствующий внешний ключ, в противном случае будет выдано сообщение об ошибке.

Кроме того, Вам необходимо знать так же имя ограничения. Если при создании таблицы имя ограничения не было указано, то следует посмотреть его имя любо через IBConsole, либо в системной таблице RDB$RELATION_CONSTRAINTS.

Например, для того, что бы посмотреть имеющиеся ограничения для таблицы people, можно выполнить следующий запрос:

```
SELECT *
FROM rdb$relation_constraints
WHERE rdb$relation_name = 'PEOPLE';
```

Так как все названия в системных таблицах приводятся к верхнему регистру, то имя таблицы набрано прописными буквами.

Синтаксис команды выглядит следующим образом:

ALTER TABLE <имя_таблицы> DROP CONSTRAINT <ограничение>

Удаление таблиц. Для удаления таблиц используется команда SQL DROP TABLE, которая имеет следующий синтаксис:

DROP TABLE <имя_таблицы>;

При удалении таблицы удаляются данные, которые в ней хранились, ее индексы и метаданные (описание) этой таблицы. Так же удаляются связанные с таблицей триггеры. Таблица может быть удалена ее создателем, администратором базы данных и, для некоторых систем, любым пользователем, имеющим привилегии root. Вы не можете удалить таблицу, поля которой используются в вычисляемых полях других таблиц, отображениях (view), ограничениях целостности данных или в хранимых процедурах. Для удаления таблицы нужно либо удалить связанные с ней объекты, либо изменить их так, что связь перестала существовать. Также невозможно удалить таблицу, используемую другими пользователями до тех пор пока они ее не освободят.

Индексы. Индекс – это механизм, увеличивающий скорость доступа к записям базы данных в ответ на запрос отбора данных по определенным условиям, а так же служащий для обеспечения уникальности данных в таблицах. Как люди используют список терминов в конце книги для быстрого доступа к необходимому содержанию, так и индекс используется сервером в качестве списка указателей на физическое расположение (адрес) строки таблицы. Таким образом, индекс представляет собой отсортированный список значений индекси

руемого поля вместе с указателями на соответствующие записи таблицы.

При выполнении запроса InterBase сервер проверяет, существуют ли индексы в указанных таблицах. Затем он определяет, что более эффективно – провести сканирование всей таблицы или использовать существующие индексы для выполнения запроса. Если сервер решает, что необходимо использовать индексы, то он осуществляет очень быстрый поиск необходимых ключевых значений в индексе и по указателям, с ними связанным, находит физическое месторасположение необходимой строки.

Индекс может быть описан как для одного столбца, так и для нескольких столбцов таблицы. Многоколоночные (составные) индексы так же могут использоваться для поиска по одной колонке в том случае, если эта колонка является первой в индексе.

В случае использования индекса время реакции на запрос существенно сокращается. Так почему не использовать индексы для всех полей таблицы? У индексов имеется два основных недостатка: первый заключается в том, что индексы увеличивают размер базы данных; второй связан с увеличением времени добавления, изменения и удаления записей таблиц, так как индекс должен обновляться при каждом изменении данных индексированных полей.

Несмотря на вышеотмеченные недостатки, преимущества индексов, связанные со скорость получения результатов запроса существенно перевешивают возможные потери места и производительности. Можно рекомендовать создавать индексы в следующих ситуациях:

- поле часто используется в условиях отбора;
- поле часто используется для соединения;
- поле часто участвует в операциях сортировки;

В некоторых ситуациях не рекомендуется использовать индексы:

- поля, которые очень редко используются в условиях поиска;
- часто изменяемые не ключевые поля;
- поля, которые имеют небольшой набор возможных значений;
- небольшие таблицы, которые целиком помещаются в несколько страниц базы данных.

Индексы могут создаваться разработчиком базы данных с помощью конструкции CREATE INDEX или автоматически сервером как часть ограничений, используемых при создании таблицы. InterBase позволяет создавать до 65536 индексов на одну таблицу.

InterBase автоматически генерирует индексы для поля или полей, которые используются в качестве первичного, уникального или внешнего ключа.

Для создания индекса используется следующая конструкция:

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]]
INDEX index ON table ( col [, col ...]);
```

где

UNIQUE – используется для создания уникального индекса, то есть не допускающего ввода повторяющихся значений для поля/полей;

ASC[ENDING] – порядок сортировки (по которому строится индекс) по возрастанию; используется по умолчанию;

DESC[ENDING] – порядок сортировки по убыванию.

Составные индексы. Главной причиной, по которой может возникнуть необходимость в создании составного индекса, является необходимость частого выполнения запросов с условиями отбора по нескольким полям. Необязательно создавать индекс, который точно соответствует используемым в запросе полям отбора, так как InterBase может использовать подмножества компонентов составного индекса для оптимизации запроса, но только при выполнении следующих условий:

- Подмножество полей, используемых в условиях отбора или сортировки запроса, являются первыми в составном индексе. Если только подмножество полей не соответствует полностью частям индекса, то InterBase не сможет использовать индекс для оптимизации такого запроса. Например, если индекс построен на базе полей A1, A2 и A3, то запрос, который использует поля A1 и A2 для отбора или сортировки, будет оптимизирован, а запрос, использующий поля A2 и A3 (или A1 и A3) не будет оптимизирован.
- Порядок, в котором поля используются в условиях должны совпадать с порядком, в котором использовались поля при создании индекса (запрос, который использует поля A2 и A1, не будет оптимизирован).

Замечание: если часто выполняемый запрос по нескольким полям использует оператор ИЛИ, то лучше создать несколько одиночных индексов для оптимизации такого запроса. Так как составной индекс построен иерархически, то запрос, выполняющий поиск по двум или более условиям, будет производиться в виде полного поиска по таблице, что не даст использовать преимущества индекса.

Улучшение производительности индексов. Индексы могут становится несбалансированными в случае многократного изменения базы данных. В этом случае может повысить производительность одним из четырех методов:

- перестроить индекс;
- перестроить избирательность индекса;
- удалить и заново создать индекс;
- зарезервировать и затем восстановить базу.

Перестройку индекса можно произвести с помощью команды

ALTER INDEX name {ACTIVE | INACTIVE};

которая включает/отключает использование индекса. При включении отключенного индекса производится его перестройка, что в случае его несбалансированности может существенно увеличить производительность. То есть, если индекс в результате множественных изменений в базе стал несбалансированным, то положение можно улучшить, отключив индекс, а затем включив его.

Так же полезно отключать индекс на время вставки большого количества записей, что существенно ускорит эту процедуру. В противном случае при каждом добавлении новой записи будет производиться изменение индекса.

Накладываются следующие ограничения на использование перестройки индекса:

- для того, чтобы перестроить индекс необходимо быть создателем индекса, администратором базы (SYSDBA), либо, в некоторых операционных системах, пользователем с привилегиями root;
- нельзя перестроить индекс, если он в данный момент используется; индекс считается используемым в случае, если есть скомпилированные запросы для выполнения; все запросы, использующие индекс, должны быть завершены для того, чтобы сделать индекс доступным;
- невозможно перестроить индекс, который создан сервером автоматически для поддержки ограничений первичных, уникальных и внешних ключей; для их изменения необходимо воспользоваться конструкцией ALTER TABLE (см. выше).
- команда ALTER INDEX не используется для удаления индекса или его изменения; для этого сначала удалите индекс (см. ниже команду DROP INDEX) а затем создайте его заново.

Изменение избирательности индекса. Для таблиц, в которых сильно изменяется количество одинаковых значений в индексных полях, периодический расчет избирательности индекса может существенно увеличить производительность.

Избирательность индекса используется оптимизатором запросов InterBase для построения плана выполнения запроса. Она отражает статистику распределения значений индексных полей. Для вычисления избирательности индекса используется следующая команда:

SET STATISTICS INDEX name;

Накладываются следующие ограничения на расчет избирательности индекса:

- для того чтобы рассчитать избирательность индекса, необходимо быть создателем индекса, администратором базы (SYSDBA), либо, в некоторых операционных системах, пользователем с привилегиями root;
- команда SET STATISTICS не перестраивает индекс; для его перестройки используйте ALTER INDEX.

Удаление индекса. Для удаления индекса используется следующая команда:

DROP INDEX name;

Для того, чтобы изменить индекс, необходимо сначала удалить старый, а затем создать новый под тем же именем.

Накладываются следующие ограничения на операцию удаления индекса:

- для того, чтобы удалить индекс, необходимо быть создателем индекса, администратором базы (SYSDBA), либо, в некоторых операционных системах, пользователем с привилегиями root;
- невозможно удалить используемый индекс до тех пор, пока он не освободиться; если попытаться удалить или изменить индекс в контексте транзакции, то результат зависит от ее свойств: в ждущей транзакции (параметр WAIT) операции ALTER INDEX и DROP INDEX ожидают освобождения нужного им индекса, в транзакции без ожидания (параметр NOWAIT) InterBase немедленно возвращает сообщение об ошибке;
- невозможно перестроить индекс, который создан сервером автоматически для поддержки ограничений первичных, уникальных и внешних ключей; для их удаления необходимо удалить сами ограничения, поля, по которым они по

строены или всю таблицу; для их изменения необходимо воспользоваться конструкцией ALTER TABLE (см. выше).

– команда ALTER INDEX не используется для удаления индекса или его изменения; для этого сначала удалите индекс (см. ниже команду DROP INDEX) а затем создайте его заново.

§ 1.3.7 Работа с данными.

Помимо команд создания таблиц, язык SQL поддерживает так же специальные команды для манипуляций данными. С помощью этих команд можно производить поиск и сортировку нужной информации, добавление новых и модификацию уже введенных в таблицы данных.

Выборка данных по запросу.

Осуществляется с помощью команды SQL SELECT, имеющий следующий синтаксис:

```
SELECT [TRANSACTION имя_транзакции]
[DISTINCT | ALL] {* | <знач> [, <знач> …]}
[INTO :переменная[, : переменная …]]
FROM <ук_таблицу> [, < ук_таблицу > …]
[WHERE <условия_поиска>]
[GROUP BY поле [COLLATE порядок_сортировки] [, col
[COLLATE порядок_сортировки] …]
[HAVING <условия_поиска>]
[UNION <select_expr> [ALL]]
[PLAN <plan_expr>]
[ORDER BY <порядок_сортировки>]
[FOR UPDATE [OF поле [, поле …]]];

<знач> = {поле [ <описание_массива>] | :переменная
| <константа> | <выражение> | <функция>
| udf ([<знач>[, <знач> …]])
| NULL | USER | RDB$DB_KEY | ?
} [COLLATE вид_сортировки] [AS alias]

<описание_массива> = [[x:]y [, [x:]y …]]

<константа> = число | 'строка' | набор_символов 'строка'

<функция> = COUNT (* | [ALL] <знач> | DISTINCT <знач>)
| SUM ([ALL] <знач> | DISTINCT <знач>)
| AVG ([ALL] <знач> | DISTINCT <знач>)
| MAX ([ALL] <знач> | DISTINCT <знач>)
```

| MIN ([ALL] <знач> | DISTINCT <знач>)
| CAST (<знач> AS <тип_данных>)
| UPPER (<знач>)
| GEN_ID (генератор, <знач>)

<ук_таблицу> = <присоединенная_таблица> | таблица | отображение | процедура[(<знач> [, <знач> …])] [alias]

<присоединенная_таблица> = < ук_таблицу> <тип_соединения> JOIN <ук_таблицу>
ON <условия_поиска> | (<присоединенная_таблица>)

<тип_соединения> = [INNER] JOIN
| {LEFT | RIGHT | FULL } [OUTER]} JOIN

<условия_поиска> = <знач> <оператор> { <знач> | (<select_one>)}
| <знач> [NOT] BETWEEN <знач> AND <знач>
| <знач> [NOT] LIKE <знач> [ESCAPE <знач>]
| <знач> [NOT] IN (<знач> [, <знач> …] | <select_list>)
| <знач> IS [NOT] NULL
| <знач> {>= | <=}
| <знач> [NOT] {= | < | >}
| {ALL | SOME | ANY} (<select_list>)
| EXISTS (<select_expr>)
| SINGULAR (<select_expr>)
| <знач> [NOT] CONTAINING <знач>
| <знач> [NOT] STARTING [WITH] <знач>
| (<условия_поиска>)
| NOT <условия_поиска>
| <условия_поиска> OR <условия_поиска>
| <условия_поиска> AND <условия_поиска>

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

<plan_expr> =
[JOIN | [SORT] [MERGE]] ({ <plan_item> | <plan_expr>}
[, { <plan_item> | <plan_expr>} …])

<plan_item> = { table | alias}
{NATURAL | INDEX (<index> [, <index> …])| ORDER <index>}

<порядок_сортировки> =
{ поле | номер_поля} [COLLATE вид_сортировки]
[ASC[ENDING] | DESC[ENDING]]
[, <порядок_сортировки> …]

где

<select_one> = SELECT по одному столбцу; должно возвращаться ровно одно значение.

<select_list> = SELECT по одному столбцу; возвращается любое количество значений.

<select_expr> = SELECT по нескольким столбцам; возвращается любое количество значений.

<выражение> = Допустимое SQL выражение, возвращающее одиночное значение.

Краткое описание основных ключевых слов команды SQL:

Таблица 6

Ключевое слово	**Описание**
TRANSACTION имя_транзакции	Имя транзакции, под управлением которой выполняется запрос; доступна только в SQL и встроенном SQL; динамический SQL не позволяет использовать такую конструкцию (IBConsole построен с использованием динамического SQL, соответственно, там нельзя использовать заранее созданные транзакции)
[DISTINCT | ALL]	Определяет данные, входящие в выборку. DISTINCT запрещает возвращение повторяющихся значений; ALL (по умолчанию) разрешает.
{* | знач [, знач …]}	Звездочка (*) возвращает все поля из указанной таблицы, а знач [, знач…] возвращает указанные поля, значения и выражения.
INTO :переменная[, : переменная …]	Запрос, возвращающий одну запись; только для встроенного SQL, указывает список локальных переменных, в которые записывается возвращенные запросом значения;
FROM < ук_таблицу> [,<ук_таблицу> …]	Список таблиц, отображений и хранимых процедур, из которых извлекаются данные; список может включать в себя соединения, которые могут быть вложенными.
процедура	Имя существующей хранимой процедуры, которая может возвращать набор данных (курсор)
alies	Краткое вспомогательное имя для таблицы, отображения, процедуры или поля запроса; в случае описания в качестве имени в пара

Ключевое слово	Описание
	метре FROM может использоваться для ссылки на соответствующий объект.
присоединен-ная_таблица	Таблица, участвующая в соединении.
тип_соединения	По умолчанию внутреннее соединение (INNER)
WHERE <усло-вия_поиска>	Определяет условия, которые ограничивают количество записей, возвращаемое запросом. Может содержать другой запрос, называемый подзапросом
GROUP BY поле, [поле]…	Группирует результат запроса по значениям полей, указанном в этом параметре.
COLLATE поря-док_сортировки	Определяет порядок сортировки, сравнения или группировки для данных, возвращаемых запросом.
HAVING <усло-вия_поиска>	Используется вместе с GROUP BY; определяет условия по полям, участвующим в функциях агрегирования, которые ограничивает количество записей, участвующих в группировке, возвращаемых запросом.
UNION [ALL]	Объединяет две или более таблиц, которые полностью или частично совпадают по структуре; параметр ALL разрешает наличие в результирующем наборе данных совпадающих записей, иначе они объединяются в одну.
PLAN <plan_expr>	Определяет план доступа для оптимизатора InterBase (возможность ручной оптимизации запросов)
plan_item	Определяет таблицу и индексирующий метод для плана оптимизации.
ORDER BY <поря-док_сортировки>	Определяет набор полей, участвующих в сортировке либо по имени поля либо по его порядковому номеру в списке полей, возвращаемых запросом, а также порядок (по возрастанию или убыванию), в котором будут отсортированы записи запроса.
FOR UPDATE	Определяет список полей после оператора SELECT в предложение DECLARE CURSOR (создание курсора), которые мо

Ключевое слово	Описание
	гут быть изменены с помощью оператора WHERE CURRENT OF.

Команда SELECT является очень мощным средством языка SQL, и позволяет выполнять практически любую обработку данных, не связанных с изменением исходных данных. В случае, если какой-либо запрос к данным не выражается через язык SQL, можно воспользоваться хранимыми процедурами, возвращающими набор записей (см. § 1.3.11). Отображения также базируются на одиночной команде SELECT (см. § 1.3.10). Одной из особенностей языка SQL состоит в том, что он выполняет операции не над одним аргументом (применительно к базам данных над одной записью таблицы), а над их множеством, что, безусловно, может приводить как к увеличению производительности приложения, так и к более простому и понятному коду. Результатом выполнения команды SELECT всегда является набор записей, поэтому результат одной команды SQL можно использовать в качестве параметров другой команды (так называемые подзапросы).

Минимально команда SELECT может состоять из двух предложений: SELECT со списком полей и FROM с указанием таблиц(ы), источники данных для указанных в конструкции SELECT полей.

Например, команда

SELECT * FROM t1;

выберет все данные из таблицы *t1*.

Для отбора данных по условию необходимо добавить к команде конструкцию WHERE. Для отбора всех записей из таблицы *t1*, у которых поле *name* равно значение ‘Иванов’, необходимо выполнить следующий запрос:

```
SELECT *
FROM t1
WHERE name = ‘Иванов’;
```

Следует заметить, что используемый в предложении SELECT символ ‘*’ обеспечивает возвращение всех полей из таблицы (или таблиц в случае соединения или декартова произведения (см. ниже)).

Предложение WHERE позволяет ограничивать количество строк, возвращаемых запросом (так же может использоваться и для объединения таблиц, см. ниже, однако такая практика не рекомендуется).

После ключевого слова WHERE указываются условия поиска. Они получили такое название, потому что конструкция WHERE часто используется для поиска определенных данных. При указании условий отбора запрос работает следующим образом: для каждой строки

Таблица *t1*

id	name
1	Иванов
2	Петров
3	Сидоров

Таблица *t2*

id_t1	t_type
1	Оператор
2	Программист
3	Инженер

запроса проверяется условия поиска и если они истинны, то это строка попадает в результирующий набор записей, в противном случае она из него исключается. Следует помнить, что любое логическое выражение может иметь три значения: истина, ложь и неизвестное значение (null). Последнее возможно в том случае, если один из операндов выражения имеет пустое значение (также null). В качестве условий поиска могут задаваться любые допустимые SQL выражения, возвращающие логический тип данных. Подробнее о сложных условиях поиска см. ниже в этом разделе.

При перечислении нескольких таблиц в конструкции FROM выполняется операция декартова произведения над множествами записей, содержащихся в указанных таблицах. В результате получается множество записей, обладающее следующими свойствами:

Степень (кол-во полей) = $\sum_{i=1}^{n}$ степень i-таблицы в конструкции FROM

Мощность (кол-во записей) = $\prod_{i=1}^{n}$ мощность i-таблицы в конструкции FROM

Из этого следует, что результирующее отношение может иметь очень большие размеры. Поэтому на практике применяют ограниченный вариант такой операции, именуемый соединением, то есть ограничивают количество столбцов и записей с помощью дополнительных условий.

Например, пусть имеются таблицы *t1* и *t2*:

Как видно из данных, между таблицами может быть установлена связь по полям *id* → *id_t1*. Однако, если мы выполним запрос в виде

```
SELECT * FROM t1, t2;
```

то получим следующий результат:

id	name	id_t1	t_type
1	Иванов	1	Оператор
1	Иванов	2	Программист
1	Иванов	3	Инженер
2	Петров	1	Оператор
2	Петров	2	Программист

id	name	id_t1	t_type
2	Петров	3	Инженер
3	Сидоров	1	Оператор
3	Сидоров	2	Программист
3	Сидоров	3	Инженер

Для обеспечения связи между таблицами можно как одним из вариантов воспользоваться условием в конструкции WHERE (другой вариант смотрим ниже):

```
SELECT * FROM t1, t2
WHERE id=id_t1;
```

В результате мы получаем гораздо более приемлемый результат:

id	name	id_t1	t_type
1	Иванов	1	Оператор
2	Петров	2	Программист
3	Сидоров	3	Инженер

Такой вариант отбора и называется соединением.

Использование модификатора DISTINCT в команде SELECT позволяет избавиться от повторяющихся записей. Например, пусть запрос

```
SELECT id, name FROM t_dir WHERE s_type=10;
```

возвращает следующий набор записей:

id	name
1	отдел 1
1	отдел 1
2	отдел 1
3	отдел 2
3	отдел 2
3	отдел 5

При добавлении DISTINCT к вышеописанной команде из полученного набора записей будут удалены повторяющиеся строки:

```
SELECT DISTINCT id, name FROM t_dir WHERE s_type=10;
```

id	name
1	отдел 1
2	отдел 1
3	отдел 2
3	отдел 5

В случае, если в соединяемых таблицах имеются поля с одинаковыми именами, то для разрешения обращения к таким полям можно воспользоваться псевдонимами. Например, при соединении таблиц

people и *name_dir*, имеющих одинаковые поля *id*, можно выполнить следующую команду:

```
SELECT a.id, b.name
FROM people a, name_dir b
WHERE a.id_name=b.id
ORDER BY b.name;
```

При этом *a* будет псевдонимом (коротким названием) для таблицы *people*, а *b* – соответственно для *name_dir*. Так же бывает полезно создавать псевдонимы для полей. Такая необходимость возникает тогда, когда необходимо, что бы в результирующем наборе записей поля называлось по другому:

```
SELECT a.id, b.name AS surname
FROM people a, name_dir b
WHERE a.id_name=b.id
```

В приведенном примере в результате выполнения запроса базовое поле name будет называться surname (фамилия).

В случае соединения таблиц, имеющих поля с одинаковым именем при создании результирующего набора система автоматически переименовывает такие поля, что бывает не всегда удобно. В этом случае также необходимо воспользоваться псевдонимом:

```
SELECT a.id, c.name AS firstname, b.name AS surname
FROM people a, name_dir b, name_dir c
WHERE a.id_name=b.id and a.id_fname=c.id
```

Для выполнения операции соединения желательно использовать не указания условий в разделе WHERE, а специальной конструкцией языка SQL – JOIN (соединение). С точки зрения особенностей выполнения использование конструкции JOIN не дает какого-либо преимущества по быстродействию и т. д., однако ее использование позволяет строить логически более стройные и понятные запросы, так как в разделе WHERE остаются только условия отбора, а операторы соединения вынесены в раздел FROM.

Interbase поддерживает два типа соединений: внешнее и внутреннее. Внутреннее соединение основано на условии соединения и возвращает только те записи из таблиц, для которых данное условие выполняется. Существует три типа внутренних соединений:

- экви-соединения: основаны на равенстве значений полей, участвующих в соединении; удобны для использования в запросе связей между таблицами;
- соединения, использующие условия, отличное от равенства полей, участвующих в соединении; редко используемый тип соединения;

– возвратные соединения (или самосоединения): в соединении используются поля из одной таблицы

Внешние соединения возвращают не только те записи, которые удовлетворяют условию соединения, но и все остальные. То есть для одной из таблиц, участвующих в соединении, возвращаются все записи, в то время как для второй возвращаются либо те записи, которые соответствуют условию соединения, либо, если условие не выполняется, пустые строки.

Чаще всего, конечно, используются внутренние соединения, так как они позволяют легко отобразить связи между таблицами. В то же время внешние соединения позволяют выбрать записи, отвечающие условию соединения в окружении записей, не отвечающих ему.

Таблицы всегда соединяются по каким-то полям. Естественно, эти поля должны быть как минимум совместимы по типу данных и иметь одинаковое значение. То есть нельзя, например, выполнить соединения по полям, одно из которых имеет целый тип, а второе – строковый. Так же логически не имеет смысла соединение между полями, в одном из которых хранится, например, идентификатор человека, а во втором – идентификатор товара, хотя формально, если они имеют совместимые типы данных, такое соединение можно выполнить.

Достаточно общим и надежным критерием для выбора полей, участвующих в соединении, является их участие во внешнем ключе на стороне одной из таблиц и, соответственно, в первичном или уникальном ключе – на стороне другой таблицы.

Поля, имеющие типы INTEGER, DECIMAL, NUMERIC, FLOAT и DOUBLE PRECISION могут участвовать в соединении между собой, так все эти типы являются числовыми и автоматически приводятся друг к другу сервером InterBase. Строковые типы, такие как CHAR и VARCHAR, могут участвовать в соединении только с другими строковыми типами, за исключением тех случаев, когда в них хранится строковое представление числа (тогда они могут соединяться с числовыми полям). Так же при соединении может использоваться функция явного приведения типа CAST(), что позволяет устанавливать соединение по полям, имеющим не совместимый тип.

Если в одном из соединяемых полей в какой-либо строке храниться значение NULL, то в случае выполнения операции соединения такая строка не будет включена в результирующий набор записей, за исключением случаев выполнения внешних соединений.

Рассмотрим примеры запросов, использующих различные варианты внутренних соединений.

Наиболее распространенной формой соединения является экви-соединения. Например, пусть у нас есть две таблицы: *employees*(*id*, *id_a*, *f_name*, *s_name*) (сотрудники) и *appointments*(*id*, *name*, *hours*) (должности), которые связаны по полям *id_a* и *id* соответственно. Если мы хотим выбрать список сотрудников и количество часов, которое они в соответствии с занимаемой должностью должны отработать в течение месяца, то соответствующий запрос будет выглядеть следующим образом:

```
SELECT a.id, a.s_name, a.f_name, b.hours
FROM people a INNER JOIN appointments b ON a.id_a=b.id;
```

Как видно из примера, мы соединяем таблицы *people* и *appointments*, при этом условие соединения указано после предложения ON. Таблица *people* называется левой, а таблица *appointments* правой, что важно для случая внешних соединений, которые будут рассмотрены ниже. В результате выполнения такого запроса мы получим список сотрудников, включающий в себя идентификатор, фамилию и имя, а также количество часов, которые необходимо этому сотруднику отрабатывать ежемесячно.

Аналогичный результат получился бы, если бы мы просто перечислили список таблиц, а условие соединения перенесли бы в предложение WHERE. В частности, пример соединении таблиц с обеспечением связи между ними, приведенный выше, при использовании конструкции JOIN, будет выглядеть следующим образом:

id	name	age
1	Иванов	20
2	Петров	18
3	Сидоров	35
4	Махов	32
5	Цветов	28

Таблица students

```
SELECT *
FROM t1 a INNER JOIN t2 b ON a.id=b.id_t1;
```

Возможно также использование разным образом вложенных соединений. Так как конструкция JOIN представляет собой такой же набор записей, как и таблица, то к одному соединению можно применять дополнительную операцию соединения, а псевдонимы (см. выше) позволяют разрешить возможные конфликты имен. Пример использования вложенных соединений (исходный вариант запроса приведен выше):

```
SELECT a.id, c.name AS firstname, b.name AS surname
```

```
FROM (people a INNER JOIN name_dir b ON a.id_name=b.id)
        INNER JOIN name_dir c ON a.id_fname=c.id;
```

В качестве условия соединения может использоваться не только равенство, но и любое допустимое SQL выражение, возвращающее логическое значение. Однако такие соединения используются существенно реже.

Рассмотрим такие соединения на примере. Пусть у нас имеется две таблицы, *students*(*id*, *name*, *age*) и *masters*(*id*, *name*, *age*), в которых храниться информация о студентах и преподавателях (номер, фамилия и инициалы и возраст). Мы хотим выбрать тех студентов, у которых возраст больше либо равен возрасту преподавателей, вместе с преподавателями, которые моложе этих студентов.

Такой запрос можно записать следующим образом:

```
SELECT a.name, b.name AS m_name, a.age, b.age as m_age
FROM students a INNER JOIN masters b ON a.age >= b.age
```

Если в исходных таблицах у нас хранились следующие данные:

id	name	age
1	Венедиктов	52
2	Мусахранов	26
3	Бранзало	30
4	Сузикова	33
5	Макаров	42

Таблица masters

Сидоров	Сузикова	35	33
Махов	Мусахранов	32	26
Махов	Бранзало	32	30
Цветов	Мусахранов	28	26

то результат выполнения запроса будет выглядеть следующим образом:

name	m_name	age	m_age
Сидоров	Мусахранов	35	26
Сидоров	Бранзало	35	30

Как мы видим из только что приведенного примера, в результате выполнения внутреннего соединения записи, не соответствующие условию соединения, не попадают в конечный набор записей. Если же необходимо, что бы все записи из одной таблицы попадали в результирующий набор, нужно использовать внешнее соединение. Например, если мы желаем вывести список всех студентов, а не толь

ко тех, которые старше некоторых преподавателей (при этом поля со сведениями о младших преподавателях должны быть пустыми), то мы можем выполнить следующий запрос:

```
SELECT a.name, b.name AS m_name, a.age, b.age as m_age
FROM students a LEFT JOIN masters b ON a.age >= b.age
```

Результат выполнения такого запроса приведен в таблице:

name	m_name	age	m_age
Иванов	null	20	null
Петров	null	18	null
Сидоров	Мусахранов	35	26
Сидоров	Бранзало	35	30
Сидоров	Сузикова	35	33
Махов	Мусахранов	32	26
Махов	Бранзало	32	30
Цветов	Мусахранов	28	26

Соответственно, при необходимости вывести все записи с преподавателями, нужно использовать правое соединение (RIGHT JOIN).

Если же необходимо, что бы в результирующий набор попали все записи из обоих таблиц, то в этом случае используется полное соединение (FULL JOIN). При этом условие соединение полностью игнорируется, а результат аналогичен декартову произведению, т. е. в этом случае можно просто перечислить таблицы через запятую в предложении FROM.

Группировка и агрегирующие функции. Для анализа содержимого различных таблиц бывает удобно представить информацию в сжатом виде, сгруппировав ее по каким-либо признакам и рассчитав для этих групп статистические данные, которые представлены в InterBase так называемыми агрегирующими функциями. Для выполнения таких задач в языке SQL предусмотрена конструкция GROUP BY.

В InterBase допустимо использовать следующие агрегирующие функции:

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

Функция	Описание
COUNT (* | [ALL] <знач> | DISTINCT <знач>)	* – возвращает количество строк в указанной в запросе таблице, включая

Функция	Описание
	строки с пустыми полями; ALL – подсчитывает количество не пустых значений указанного поля; DISTINCT – подсчитывает количество уникальных не пустых значений указанного поля; <знач> – имя поля или выражение.
SUM ([ALL] <знач> | DISTINCT <знач>)	ALL – подсчитывает общую сумму значений указанного поля; DISTINCT – подсчитывает общую сумму значений указанного поля, предварительно удалив повторяющиеся значения; <знач> – имя поля, константа, выражение, не агрегирующая функция, или UDF, возвращающая числовое значение.
AVG ([ALL] <знач> | DISTINCT <знач>)	ALL – подсчитывает среднее всех значений указанного поля; DISTINCT – подсчитывает среднее всех значений указанного поля, предварительно удалив повторяющиеся значения; <знач> – имя поля или выражение, возвращающее числовой тип данных.
MAX ([ALL] <знач> | DISTINCT <знач>)	ALL – находит максимальное значение для указанного поля; DISTINCT – находит максимальное значение для указанного поля, предварительно удалив повторяющиеся значения; <знач> – имя поля, константа, выражение, не агрегирующая функция, или UDF, возвращающая числовое значение.
MIN ([ALL] <знач> | DISTINCT <знач>)	ALL – находит минимальное значение для указанного поля; DISTINCT – находит минимальное значение для указанного поля, предварительно удалив повторяющиеся

Функция	Описание
	значения; <знач> – имя поля, константа, выражение, не агрегирующая функция, или UDF, возвращающая числовое значение.

COUNT подсчитывает количество строк, удовлетворяющих условию запроса Может использоваться для отображений (view), соединений (join) и таблиц.

В качестве источника для агрегирующих функций SUM и AVG могут выступать только числовые значения.

Если значение поля или выражения неизвестно или является пустым (null), то это значение автоматически игнорируется при расчете значения функции AVG. Это делается для предотвращения искажений, вызванных незначащими данными.

При выполнении функции MIN или MAX из поиска исключаются поля, содержащие пустые (NULL) значения.

Если количество записей передаваемой агрегирующей функции равно нулю, то она возвращает пустое (NULL) значение.

Рассмотрим пример выполнения групповых операций на основе предыдущих примеров. Пусть нас интересует информация не о том, какие именно преподавателю младше студентов, а о их количестве. В этом случае необходимо прибегнуть к операции группировки:

```
SELECT a.name, count(b.id) AS m_younger
FROM students a LEFT JOIN masters b ON a.age >= b.age
GROUP BY a.name;
```

Результат выполнения данного запроса будет выглядеть следующим образом:

name	m_younger
Иванов	0
Махов	2
Петров	0
Сидоров	3
Цветов	1

Заметьте, что по умолчанию поля, участвующие в группировке, всегда сортируются по возрастанию. Если же необходим иной порядок сортировки, то необходимо указать после предложения GROUP BY предложение ORDER BY с требуемым порядком сортировки.

При выполнении операции группировки необходимо помнить о следующих условиях ее применения:

- каждое поле, по которому производится группировка, должно присутствовать в предложении SELECT;
- в предложении GROUP BY нельзя указывать математическое выражение, агрегирующую или описанную пользователем (UDF) функцию;
- нельзя использовать подзапросы, ссылающиеся на отображения (view), содержащие в описании предложения GROUP BY или HAVING;
- для каждого предложения SELECT в запросе (включая подзапросы), может только одно предложение GROUP BY;

Агрегирующие функции можно использовать и без предложения GROUP BY, в этом случае все поля, перечисленные в предложении SELECT должны участвовать в агрегирующих операциях; при этом набор записей, возвращаемых командой SELECT, считается одной группой.

Так же как и в обычных запросах предложение WHERE ограничивает количество возвращаемых записей, так и при использовании операции GROUP BY можно ограничить количество записей с использованием операции HAVING. В качестве условий предложения HAVING могут использоваться те же конструкции, что и для WHERE, со следующими ограничениями:

- обычно каждое условие связанно с соответствующим агрегированным полем в предложении SELECT;
- в подзапросе, используемом в условии к предложению HAVING не могут использоваться таблицы или отображения, присутствующие в конструкции FROM основного запроса;
- связанные подзапросы не могут использоваться в предложении HAVING;

Так же HAVING может использоваться и без предложения GROUP BY. В этом случае вся выборка трактуется как одна группа, а каждое поле, перечисленное в предложении SELECT – как агрегированное.

Не стоит так же забывать, что отбор по условиям HAVING выполняется после операции группировки и вычисления агрегированных значений, поэтому не стоит использовать операцию HAVING к полям, не входящим в операции агрегирования, так как использование по таким полям отбора с помощью конструкции WHERE даст тот же результат, но будет производится до операции группировки, что безусловно ускорит выполнение запроса.

Пример использования предложения HAVING:

```
SELECT a.name, count(b.id) AS m_younger
FROM students a LEFT JOIN masters b ON a.age >= b.age
GROUP BY a.name
HAVING count(b.id) > 1;
```

Результат выполнения этого запроса будет выглядеть следующим образом:

name	m_younger
Махов	2
Сидоров	3

Иногда возникает необходимость выполнить объединение таблиц или наборов записей. Такая необходимость определяется тем, что в таблицах хранится сходная информация. Для выполнения таких действий в языке SQL используется предложение UNION. Оно позволяет взять данные из одной таблицы (или набора записей) и добавить к ним записи из другой таблицы, при необходимости удалив повторяющиеся строки.

Например, нам нужно в одном наборе записей представить список всех людей, на данное время работающих или учащихся в институте. Однако информация о студентах и преподавателях находится в разных таблицах, поэтому что бы объединить ее необходимо в конструкцию SELECT добавить предложение UNION:

```
SELECT a.id, a.name
FROM students a
UNION
SELECT b.id, ,b.name
FROM masters b
```

Данный запрос вернет список всех студентов и сотрудников института, при этом дублирующиеся записи будут удалены (они возможны в том случае, если преподаватель был раньше студентом этого же учебного заведения). Иногда бывает необходимо оставить дублирующие записи, в этом случае необходимо добавить конструкцию ALL:

```
SELECT a.id, a.name
FROM students a
UNION ALL
SELECT b.id, ,b.name
FROM masters b
```

Если нам необходим список лиц, в данное время работающих или учащихся в институте, то мы можем добавить условия отбора:

```
SELECT a.id, a.name
```

```
FROM students a
WHERE a.dog is null
UNION ALL
SELECT b.id, ,b.name
FROM masters b
WHERE b.dod is null
```

Здесь dog и dod соответственно поля, содержащие сведения о дате окончания института или увольнения с должности.

Часто предложения UNION используется для выполнения операции группировки, при этом группировка не может выполняться для всего набора записей, а только для каждого отдельной конструкции SELECT.

Пусть сведения о выплате стипендий студентам хранятся в таблице *scholarship*(*id_st*, *s_date*, *summa*): идентификатор студента, дата выдачи стипендии, выданная сумма; а сведения о зарплате преподавателей – в таблице *salary*(*id_m*, *s_date*, *summa, taxes*): идентификатор преподавателя, дата выдачи заработной платы, выданная сумма, уплаченные налоги. Для налоговой инспекции нам необходимо выдать сведения о полученных доходах за 2000 год, а так же указать сведения о уплаченных за этот же период налогах (студенты налогов не платят). Для выполнения этой задачи необходимо выполнить следующий запрос:

```
SELECT a.name, sum(b.summa), sum(0)
FROM students a left join scholarship b on (a.id=b.id_st) and
      (EXTRACT(YEAR from b.s_date)=2000)
GROUP BY a.name
UNION
SELECT c.name, sum(d.summa), sum(d.taxes)
FROM masters c left join salary d on (c.id=d.id_m) and
      (EXTRACT(YEAR from d.s_date)=2000)
GROUP BY c.name
ORDER BY 1;
```

Заметьте, что в данном случае используется левое внешнее соединение для того, что бы получить данные по всем студентам и преподавателям. В случае внутреннего соединения мы бы получили сведения только о студентах и сотрудниках, которым была начислена стипендия или заработная плата в 2000 году.

При использовании объединения группировка всегда применяется к каждой конструкции SELECT, а сортировка – ко всему запросу.

Добавление записей.

Для добавления записей в таблицу используется команда INSERT, имеющая следующий синтаксис:

INSERT [TRANSACTION имя_транзакции] INTO <объект> [(поле [, поле …])]
{VALUES(<знач> [, <знач> …]) | <select_expr>};

<объект> = имя_таблиц | имя_отображения

<знач> = {:переменная | <константа> | <выражение>
| <функция> | udf([<знач>[, <знач> …]])
| NULL | USER | RDB$DB_KEY | ?
} [COLLATE collation]

<функция> = CAST (<знач> AS <тип_данных>)
| UPPER(<знач>)
| GEN_ID(имя_генератора, <знач>)

Здесь:

UPPER – функция, преобразующая текст к верхнему регистру;

<выражение> – допустимое SQL выражение, возвращающее одиночное значение;

<select_expr> – команда SELECT, которая возвращает ноль или более записей и количество полей в которой (и их тип) совпадает с количеством значений, которое будет вставляться; в таблицу будут вставлены данные, возвращаемые командой SELECT. Если SELECT возвращает несколько строк, то, соответственно, в таблицу будет вставлено такое же количество записей;

TRANSACTION имя_транзакции – определяет открытую транзакцию, в контексте которой будет происходит добавление записей; транзакция должна допускать запись данных;

INTO <объект> – определяет объект (таблицу или отображение), в который будет производится добавление записей;

VALUES(<знач> [, <знач> …]) – список значений, которые будут добавляться; значения должны быть перечислены в том же порядке, как и поля назначения.

Команда INSERT контролируется сервером на предмет допустимых разрешений на добавления данных в таблицу.

Значения вставляются в таблицу (или отображение), в котором они описаны в таблице, за исключением тех случаев, когда порядок полей указан явно после названия объекта. Если список полей являет

ся подмножеством всех полей таблицы, то в оставшиеся поля будут внесены значения по умолчанию или пустое значение (NULL).

Если список полей назначения отсутствует, то необходимо указывать значения для всех полей объекта.

Для вставки одиночной записи можно просто указать значения для всех полей.

Для вставки нескольких записей нужно указать *select_expr*, который возвращает уже существующие данные из других таблиц в точном соответствии с порядком и типом полей добавляемой таблицы.

Внимание! Разрешено производить вставку из той же таблицы, в которую добавляются данные, однако такая практика не рекомендуется, так как может приводить к бесконечной вставки данных. Для того, что бы избежать этого, необходимо гарантировать, что вновь вставляемые данные не соответствуют условиям отбора *select_expr*.

Изменение данных.

Для изменения существующих данных используется команда UPDATE, имеющая следующий синтаксис:

Форма SQL:

```
UPDATE [TRANSACTION имя_транзакции] <объект>
    SET поле = <знач> [, поле = <знач> …]
[WHERE <условия_поиска>;
```

Форма DSQL и isql:

```
UPDATE { table | view}
    SET поле = <знач> [, col = <знач> …]
[WHERE <условия_поиска>]

<объект> = имя_таблиц | имя_отображения

<знач> = {col [ <оп_массива>]
| :переменная
| <константа>
| <выражение>
| <функция>
| udf([ <знач> [, <знач> …]])
| NULL
| USER
| ?}
[COLLATE collation]

<функция> = CAST ( <знач> AS <тип_данных>)
| UPPER (<знач>)
| GEN_ID (имя_генератора, <знач>)
```

Описание <условия_поиска> находится в § 1.3.6.

В таблице 7 приводятся основные аргументы команды с их описанием.

Команда UPDATE изменяет одну или несколько строк таблицы или отображения. Выполнение команды осуществляется в соответствии с правами доступа, описанными для изменяемого объекта.

Для команды изменения с условиями необходимо указать условия, согласно которым будут отбираться записи для изменения. Команда UPDATE с условиями не может применяться для изменения полей-массивов.

Внимание! Без условия команда UPDATE изменяет **все** поля в таблице.

Таблица 7

Аргумент	**Описание**
TRANSACTION имя_транзакции	Имя открытой транзакции, в контексте которой производится изменение данных; транзакция должна допускать запись данных.
SET поле = <знач>	Определяет поле, которое будет менять и новое значение для него.
WHERE <условия_поиска>	Изменение с условием; меняются только записи, которые удовлетворяют условиям поиска.

Удаление записей.

Для удаления записей из таблицы используется команда DELETE, имеющая следующий синтаксис:

DELETE FROM <объект> [WHERE <условия_поиска>];

<объект> = имя_таблиц | имя_отображения

<условия_поиска> – см. описание команды SELECT выше в этом параграфе.

Команда DELETE определяет одну или несколько записей для удаления из таблицы или обновляемого отображения. Выполнение команды осуществляется в соответствии с правами доступа, описанными для изменяемого объекта.

Для удаления части записей из всех, имеющихся в таблице, необходимо использовать команду удаления с условиями.

Внимание! Команда удаления без условий удаляет **все** записи из таблицы.

§ 1.3.8 Триггеры; генераторы

Генераторы

Генератор – это объект базы данных InterBase, который позволяет создавать уникальный последовательный ряд чисел, которые автоматически вставляются в базу данных при добавлении или изменении записи. Обычно генераторы используются для получения уникального значения, которое вставляется в поле таблицы, играющего роль первичного (или уникального ключа) и участвует в межтабличных связях на стороне один.

В базе данных может быть описано любое количество генераторов при условии, что каждый генератор имеет уникальное имя. Генератор является глобальной переменной для базы данных, в которой он описан. Любая транзакция, которая активирует генератор, может использовать или изменить значение генератора. InterBase поддерживает уникальное значение генератора для всех транзакций.

Для создания генератора используется следующая команда:

CREATE GENERATOR name;

Созданному генератору присваивается значение ноль. Если необходимо присвоить генератору какое-либо другое значение, то необходимо выполнить команду SET GENERATOR (см. ниже).

Внимание! После того, как генератор описан, он не может быть удален из базы данных с помощью специальной команды. Для удаления генератора удалите соответствующую запись из системной таблицы RDB$GENERATORS, например:

DELETE FROM RDB$GENERATORS
WHERE RDB$GENERATORS_NAME = ‘EMP_NO’;

Изменение значения генератора. Для изменения значения генератора воспользуйтесь следующей командой:

SET GENERATOR NAME TO int;

Новое значение генератора (параметр int) должно быть целым числом из диапазона от -2^{63} до $2^{63}-1$.

Внимание! Не изменяйте значение генератора, если полностью не уверены, что не возникнут повторяющиеся значения при добавлении записей. То есть если генератор используется для автоматического заполнения поля, являющегося первичным или уникальным ключом и его значение будет изменено таким образом, что при его использовании будут генерироваться значения, уже имеющиеся в таблице, то все последующие операции вставки и изменения будут прерваны с ошибкой “Duplicate key” (повторяющийся ключ).

В базах данных только для чтения генераторы могут возвращать свое значение, но не могут изменяться.

Использование генератора. После создания генератора необходимо определить правила его использования. Для того, что бы получить значение генератора, необходимо вызвать функцию GEN_ID().GEN_ID() принимает два аргумента в качестве параметров: имя генератора (который к этому моменту должен быть уже создан) и шаг, который определяет, насколько значение генератора должно измениться. Шаг может быть как положительным, так и отрицательным. Эта функция может быть вызвана из триггера, хранимой процедуры или команды SQL, обычно при выполнении команд модификации данных таблицы. Синтаксис функции выглядит следующим образом:

GEN_ID(имя_генератора, шаг);

Пример вызова GEN_ID из команды SQL:

INSERT INTO SALES (PO_NUMBER) ЗНАЧUES (GEN_ID(G, 1));

Для того, что бы исключить автоматическую вставку значения генератора в поле таблице, измените или удалите триггер, хранимую процедуру или команду SQL, так, что бы она более не вызывала функцию GEN_ID.

Информацию по использованию генераторов в триггерах см. ниже, а в хранимых процедурах – см. § 1.3.10

Триггеры.

Триггер – это неявная процедура, связанная с таблицей или отображением (view), которая автоматически вызывается в случае добавления, изменения или удаления записи. Триггеры никогда не вызываются явно. Взамен этого, когда приложение или пользователь выполняют SQL команду, модифицирующую данные таблицы, ассоциированные с ней триггеры выполняются (срабатывают) автоматически.

Триггеры могут использовать *исключения* (см. § 1.3.12) для обработки ошибки. Когда исключительная ситуация вызывается триггером, то она возвращает сообщение об ошибке, прерывает выполнения триггера и отменяет вся изменения, сделанные триггером до этого момента, если только исключение не обработано внутри триггера оператором WHEN.

Триггеры позволяют:

- производить автоматическое выполнение ограничений базы, что позволяет добиться ввода только корректных данных пользователем;
- снизить издержки поддержки базы данных, так как изменения триггера приводит к автоматическим изменениям поведения приложения, которое использует соответствующую

таблицу без необходимости его повторной компиляции и компоновки;

- производить автоматическую запись изменений таблиц; приложение может хранить синхронный протокол изменений, который ведется триггером, вызываемым при модификации таблицы;
- производить автоматическую генерацию сообщений об изменениях в базе;
- автоматически преобразовывать данные, например, для преобразования текста к верхнему регистру.

Создание триггеров. Триггер создается с помощью команды CREATE TRIGGER, которая состоит из заголовка и тела. Заголовок триггера содержит:

- название триггера, уникальное для базы данных;
- имя таблицы, с которой связан триггер;
- предложения, определяющего момент срабатывания триггера.

Тело триггера состоит из:

- необязательного списка локальных переменных с описанием их типа;
- группы (блока) предложений процедурного языка InterBase, ограниченной операторами BEGIN и END; эти предложения выполняются при срабатывании триггера; группа может включать в себя другие группы, таким образом они могут образовывать много вложенных уровней.

Внимание! Так как каждое предложение в теле триггера должно быть отделено точкой с запятой, необходимо описать другой терминальный символ для всего триггера в целом. В ISQL используйте команду SET TERM для замены терминального символа, а после завершения тела триггера другую команду SET TERM, которая определяет терминальный символ обратно как точка с запятой. В некоторых средствах визуальной разработки баз данных (например, QuickDesk, см. § 1.3.15) терминаторы заменяются автоматически, поэтому нет необходимости их выполнять вышеописанные действия.

Синтаксис для создания триггера:

```
CREATE TRIGGER имя_триггера FOR {имя_таблицы | имя_отображения}
    [ACTIVE | INACTIVE]
    {BEFORE | AFTER} {DELETE | INSERT | UPDATE}
    [POSITION позиция]
```

AS <тело_триггера>

<тело_триггера>=[<список описаний переменных>] <блок>

<список описаний переменных> =

DECLARE VARIABLE имя_переменной тип_переменной;
[DECLARE VARIABLE имя_переменной тип_переменной;…]

<блок>=

BEGIN

<составной_оператор>[< составной_оператор >…]

END

<составной_оператор>={<блок>|оператор;}

Аргумент	Описание
FOR{имя_таблицы | имя_отображения}	Обязательный параметр. Установление связи триггера с объектом базы данных (таблицей или отображением (см. § 1.3.10)).
ACTIVE | INACTIVE	Необязательный параметр Определяет, будет ли действовать триггер: ACTIVE – триггер активен (по умолчанию); INACTIVE – триггер отключен.
BEFORE | AFTER	Требуемый параметр, определяющий, когда сработает триггер: BEFORE – до связанной с триггером операции; AFTER – после связанной с триггером операции.
DELETE | INSERT | UPDATE	Определяет операцию с таблицей, которая будет вызывать срабатывание триггера: DELETE – при удалении записи; INSERT – при вставке записи; UPDATE – при изменении записи;
POSITION позиция	Определяет порядок выполнения триггеров перед или после соответствующего события для таблицы. Позиция должна быть целым числом от 0 до 32767 включительно. Триггер с меньшим номером срабатывает первым. По умолчанию 0: первый в очереди для выполнения. Триггеры не обязаны следовать один за другим. Триггеры для обработки одного и того

Аргумент	Описание
	же события и с одной и той же позицией будут вызываться в алфавитном порядке.
DECLARE VARIABLE имя_переменной тип_переменной;	Описывает локальную переменную, используемую только в триггере. Каждое определение переменной должно завершаться точкой с запятой. Имя_переменной должно быть набрано латинскими символами.
оператор	Любой допустимый простой оператор процедурного языка InterBase. Любой оператор, за исключением BEGIN и END должен быть завершен точкой с запятой.

Описание процедурного языка InterBase см. в § 1.3.10.

Пример указания нового терминального символа перед созданием генератора:

```
SET TERM !!;
CREATE TRIGGER SIMPLE FOR EMPLOYEE
AFTER UPDATE AS
BEGIN
        …
END !!
SET TERM ;!!
```

В вышеприведенной последовательности команд выполняются следующие действия:

1. Текущий терминатор (;) заменяется на (!!).
2. Выполняется команда создания триггера.
3. Терминатор (!!) заполняется на обычный (;).

При выполнении команды SET TERM ее необходимо заменять текущим терминатором.

Ошибки при создании триггера. InterBase может генерировать ошибки при разборе триггера в случае некорректного использование синтаксиса. Сообщение оь ошибке выглядит следующим образом:

```
Statement failed, SQLCODE = -104
Dynamic SQL Error
-SQL error code = -104
-Token unknown - line 4, char 9
-tmp
```

С помощью ключевых слов line и char указывается строка и столбец, возле которого расположен ошибочный оператор. Счет ли

ний начинается с команды CREATE TRIGGER, а не от начала файла описания данных. Так же указывается вид ошибки, в данном случае это Token unknown – неизвестная лексема. Ошибочный оператор может быть расположен непосредственно в указанном месте или до этого места. В случае сомнений необходимо проверить всю строчку для определения причины ошибки.

Особенности описания триггеров.

Процедурный язык InterBase включает в себя все стандартные SQL команды, за исключением команд описания данных и работы с транзакциями, а также операторы, специфичные для самого процедурного языка.

Операторы процедурного языка включают в себя:

1. Операции присвоения, используемые для установки значений локальных переменных.
2. Операторы управления порядком команд, такие как IF…THEN, WHILE…DO и FOR SELECT…DO для выполнения команд ветвления и циклов.
3. Оператор EXECUTE PROCEDURE для вызова хранимой процедуры.
4. Оператор EXCEPTION для вызова исключительной ситуации и оператор WHEN для ее обработки.
5. Контекстные переменные NEW и OLD для временного хранения старых и новых значений записи таблицы. Характерны только для генераторов.
6. Генераторы для получения уникального значения для использования в выражениях. Генераторы могут использоваться как триггерах, так и в хранимых процедурах и командах SQL, хотя наиболее часто они используются именно в триггерах.

Подробное описание процедурного языка см. в § 1.3.10.

Переменные NEW и OLD.

Триггеры могут использовать две встроенных контекстных переменных NEW (новая) и OLD (старая) для обращения соответственно к новым значениям записи таблицы (при операциях вставки или изменения записи) или текущим (старым) значениям при операциях вставки или удаления записи. В случае удаления записи переменная NEW не используется. Эти переменные часто используются для сравнения значений записи до и после ее изменения.

Обращения к вышеописанным переменным происходит следующим образом:

NEW.поле

OLD.поле

где “поле” – название колонки изменяемой записи таблицы. Контекстные переменные могут использоваться точно так же как и обычные.

Новые значения полей записи могут быть изменены только в триггере, срабатывающем до действия с которым он связан (спецификатор BEFORE). В триггере срабатывающем после действия (спецификатор AFTER) значение переменной NEW можно изменить, однако это не приведет к каким-либо изменениям данных, хранимых в записи таблицы. Реальные значения полей не изменяются до действий со спецификатор “после изменений” (AFTER), поэтому триггеры со спецификатором BEFORE не смогут получить новые значения записи после выполнения операций вставки/изменения с помощью SQL запросов. Однако новые значения записи доступны триггерам со спецификатором AFTER.

Пример. Нижеописанный триггер срабатывает после изменения записи таблицы EMPLOYEE, сравнивает значение нового и старого оклада и в случае изменения вставляет запись журнал учета изменений оклада (таблица SALARY_HISTORY):

```
SET TERM !!;
CREATE TRIGGER SAVE_SALARY_CHANGE FOR EMPLOYEE
AFTER UPDATE AS
BEGIN
IF (old.salary <> new.salary) THEN
INSERT INTO SALARY_HISTORY (EMP_NO, CHANGE_DATE,
UPDATER_ID, OLD_SALARY, PERCENT_CHANGE)
VALUES (old.emp_no, 'now', USER, old.salary,
(new.salary-old.salary)*100/old.salary);
END !!
SET TERM ;!!
```

В данном примере ‘now’ возвращает текущую дату и время, USER – имя входа пользователя.

Примечание:

1. Контекстные переменные некогда не требуют подстановки двоеточия, даже в конструкциях SQL.
2. Если триггер выполняет действие, которое приводит к его повторной активизации – или приводит к срабатыванию другого триггера, который приводит к активации первого –

возникает бесконечный цикл. Поэтому очень важно для разработчика БД быть абсолютно уверенным, что такая ситуация не возникнет ни прямо, ни косвенно. Например, бесконечный цикл возникает, если триггер определен как срабатывающий на вставку записи и при его выполнении производится вставка новой записи в ту же таблицу.

Использование генераторов.

Функцию GEN_ID можно использовать таким же образом, как и переменную целого типа. Ниже приведен пример триггера, организующего счетчик для таблицы CUSTOMER:

```
SET TERM !!;
CREATE TRIGGER SET_CUST_NO FOR CUSTOMER
BEFORE INSERT AS
BEGIN
    NEW.CUST_NO = GEN_ID(CUST_NO_GEN, 1);
END !!
SET TERM ;!!
```

Внимание! Подобные триггеры должны быть всегда реализованы со спецификатором BEFORE INSERT (до вставки), так как это необходимо для изменения значения записи с помощью переменной NEW.

Изменение триггеров

Триггер может быть изменен только его создателем или администратором базы. Команда изменения триггера позволяет изменить:

- только заголовок триггера (активность триггера (включен/выключен), время срабатывания (до или после события), событие, вызывающее триггер и порядок, в котором триггер вызывается по отношения к другим подобным триггерам);
- тело триггера (все, что идет за служебным словом AS, то есть локальные переменные и операторы);
- заголовок и тело триггера одновременно.

Для изменения триггеров, автоматически созданных системой для реализации ограничений типа CHECK (проверка значений полей таблицы), используйте команду ALTER TABLE (см. § 1.3.6) для изменения самих ограничений.

Синтаксис команды изменения триггера выглядит следующим образом:

```
ALTER TRIGGER name
[ACTIVE | INACTIVE]
```

[{BEFORE | AFTER} {DELETE | INSERT | UPDATE}]
[POSITION number]
AS < trigger_body>;

Синтаксис для этой команды такой же, как для команды создания триггера (см. выше) за исключением:

- ключевое слово CREATE заменено словом ALTER;
- опущена конструкция FOR имя_таблицы, так как описываемая команда не может связать существующий триггер с другой таблицей;
- команда требует указывать только те параметры которые меняются, с исключениями, описанными ниже.

Изменение заголовка триггера. В этом случае команда требует наличия хотя бы одного изменяемого параметра после имени триггера. Все опущенные параметры остаются без изменения.

Следующий пример делает триггер SAVE_SALARY_CHANGE неактивным:

ALTER TRIGGER SAVE_SALARY_CHANGE INACTIVE;

Если меняется параметр времени срабатывания триггера (BEFORE или AFTER), то также необходимо указывать и операцию (INSERT, UPDATE или DELETE), с которой связывается триггер. Например, для активации триггера SAVE_SALARY_CHANGE и указания того, что он срабатывает до изменения записи, а не после, используется следующая команда:

```
ALTER TRIGGER SAVE_SALARY_CHANGE
    ACTIVE
    BEFORE UPDATE;
```

Изменение тела триггера. При изменение тела триггера полностью заменяется старое описание тела триггера на новое. Для изменения только тела триггера необходимо также указывать также какую либо информацию, кроме имени триггера.

Например, следующая команда изменения триггера модифицирует предварительно введенный триггер SET_CUST_NO для вставки записи в таблицу (предварительно созданную) NEW_CUSTOMERS о каждом новом клиенте:

```
SET TERM !!;
ALTER TRIGGER SET_CUST_NO
BEFORE INSERT AS
BEGIN
    new.cust_no = GEN_ID(CUST_NO_GEN, 1);
    INSERT INTO NEW_CUSTOMERS(new.cust_no, TODAY);
END !!
```

SET TERM ;!!

Здесь системная функция TODAY используется для возвращения текущей даты на сервере.

Удаление триггеров. Во время разработки БД и приложений иногда возникает необходимость удалить триггер. Для удаления триггера из базы используется следующая команда:

DROP TRIGGER имя_триггера;

К этой команды применяются следующие ограничения:

- только создатель триггера или администратор сервера БД может его удалить;
- нельзя удалить используемый триггер;
- нельзя удалить созданный системой триггер (для реализации ограничений таблицы или поля).

Триггеры и транзакции.

Триггеры работают в контексте транзакции, открытой приложением, которое вызвало срабатывание триггера. Если затем транзакция откатывается, то также откатываются все изменения, сделанные триггером.

Права доступа.

§ 1.3.9 Пример: разработка структуры базы данных

§ 1.3.10 Отображения данных

В этом параграфе описывается, что такое отображения данных (view), в каких случаях ими необходимо пользоваться, как создавать и удалять отображения и как модифицировать данные при их использовании.

Введение

Пользователям базы данных обычно требуется получить доступ не ко всем ее данным, а только к их части. Кроме того, данные, интересные индивидуальным пользователям или их группам, обычно совпадают. Отображения данных предоставляют способ, с помощью которых данные таблиц можно представить в виде набора данных, интересных пользователям или их группам.

После того, как отображение создано, с ним можно работать как с обычной таблицей. Отображение может создано на базе одной или нескольких таблиц или на базе других отображений. Хотя отображения выглядят как обычные таблицы, на самом деле данные, которые они представляют, не хранятся отдельно в базе данных. Вместо этого при обращении к базе данных сервер динамически формирует

необходимый набор данных как результат запросов, на основе которых построены соответствующие отображения. Следует понимать, что отображения не создают копии данных, содержащихся в таблицах, поэтому когда пользователь изменяет данные с помощью отображения, он изменяет данные в таблицах, на которых это отображение построено. И наоборот, если данные в таблицах, на которых базируется отображение, изменились, то это отразиться в данных, которые предоставляются этим отображением. Лучше всего представлять себе отображение, как окно или рамку, через которое просматриваются реальные данные. Описание отображение и определяет границы этой рамки.

Отображение может быть создано как:

- Вертикальное подмножество полей таблицы (привести пример).
- Горизонтальное подмножество записей таблицы (привести пример).
- Комбинацией горизонтального и вертикального подмножеств таблицы (привести пример).
- Подмножеством полей и записей нескольких соединенных таблиц (привести пример).

Достоинства отображений.

Основными достоинствами отображений являются:

1. Упрощение доступа к данным. Отображения позволят брать подмножество данных одной или нескольких таблиц в качестве основы других запросов без необходимости повторять часто используемые операторы SQL для возвращения того же набора данных.
2. Разнообразить доступ к данным. Отображения предоставляют способ, позволяющий удовлетворить пользователей с разнообразными требованиями, навыками и интересами. Таким образом пользователь приобретает возможность сосредоточится на необходимой ему информации, не тратя время на просмотр не нужных ему дынных.
3. Независимость данных. Отображения ограждают конечных пользователей от изменений, вносимых в структуру базы данных. Например, если разработчик БД решил разбить одну таблицу на две связанных, то отображение позволит сделать так, что для конечных пользователей две новых таблицы будут представлены как одна старая.
4. Ограничения доступа. Это достигается путем запрещения доступа к отдельным порциям данных. Например, пользова

тель может просмотреть информация о должностях сотрудников, но не может получить информацию об их окладе.

Создание отображений

Для создания отображений используется следующая команда:

```
CREATE VIEW имя_отображения [(поле[, поле …])]
AS <select> [WITH CHECK OPTION];
```

Пользователь, который создал отображение, имеет полные права на него, включая возможность предоставлять доступ к этому отображению другим пользователям, триггерам и хранимым процедурам. Пользователю можно предоставить доступ к отображению, даже если у него нет прав доступа к таблицам, на базе которых это отображение построено.

Внимание! Невозможно создать отображение на базе хранимой процедуры.

Список полей для отображения не является обязательным, за исключением случая наличия в отображении вычисляемых полей. В случае описания списка полей они соответствуют полям, возвращаемым запросом и должны так же совпадать по количеству. Таким образом необходимо либо описывать все поля, либо не описывать их вовсе.

Название поля должно быть уникальным для данного отображения. Если список полей не указан, то будут использованы названия полей, возвращаемых запросом.

SELECT в отображении выполняет следующие функции:

1. Описывает поля для включения из базовой таблицы.
2. Определяет таблицу – источник данных.
3. Определяет при необходимости условия отбора данных из таблицы.
4. Если используется параметр WITH CHECK OPTION, то он предотвращает добавление или изменение данных, которые не соответствуют условиям, указанным в запросе (только для изменяемого отображения).

Внимание! При создании отображений в запросе не может быть указан параметр ORDER BY (т. е. невозможно задать сортировку).

Обновляемые и необновляемые отображения.

Изменения, вносимые в отображения, могут передаваться в базовые таблицы только при наличие нескольких условий. В случае, если эти условия выполняются, то отображение называется обновляемым, в противном случае оно необновляемое и с его помощью нельзя ввести данные в таблицы, на котором оно основано.

Внимание! Термины “обновляемое” и “необновляемое” относятся только к данным отображений, но не к их структуре. Что бы изменить метаданные, относящиеся к отображению, необходимо сначала удалить его (см. ниже), а затем создать заново с необходимыми изменениями.

Отображение является обновляемым только при одновременном выполнении следующих условий:

- отображение является подмножеством одной таблицы или одного изменяемого отображения;
- все поля базовой таблицы, исключенные из отображения, допускают пустые значения;
- SELECT, используемый в качестве источника данных для отображения, не должен включать в себя подзапросы, предикат DISTINCT, операторы GROUP BY и HAVING, агрегирующие функции, соединенные таблицы, описанные пользователем функции или хранимые процедуры.

Если хотя бы одно из вышеперечисленных условий не выполняется, то отображение будет необновляемым.

Примечание. Необновляемое отображение можно сделать обновляемым с помощью соответствующих триггеров (см. § 1.3.8)

Для создания отображения необходимы следующие права:

- для создания необновляемых отображений необходимо право SELECT на все таблицы, на основе которых создается отображение;
- для создания обновляемых отображений необходимо право ALL на все базовые таблицы;

Удаление отображений

Команда удаления отображения имеет очень простой синтаксис:

DROP VIEW имя_отображения;

Данная команда не приводит к удалению каких-либо данных из базовых таблиц.

Для удаления отображения необходимо выполнение следующих условий:

- быть создателем отображения или администратором сервера;
- отображение не должно использоваться другими объектами базы данных; перед удалением отображения сначала необходимо удалить связанные с ним объекты (другие отображения, хранимые процедуры, ограничения типа CHECK).

Примечание. Невозможно непосредственно исправить отображение. Для того, что бы изменить его, сначала необходимо удалить его, а затем создать заново с необходимыми изменениями.

§ 1.3.11 Встроенные процедуры

В этом параграфе описываются следующие моменты, связанные с встроенными (хранимыми) процедурами:

1. Каким образом создавать, изменять и удалять хранимые процедуры.
2. Процедурный язык InterBase, используемый также в триггерах.
3. Использование хранимых процедур.
4. Обработка ошибок.

Введение

Вызов хранимых процедур

Существуют два типа хранимых процедур:

- Процедура типа SELECT, которую приложение может использовать как таблицу или отображение в качестве источника данных. Такая процедура должна быть создана так, что бы возвращать набор значений (курсор) или код ошибки.
- Исполняемая процедура, которую приложение может вызывать непосредственно, использую команду EXECUTE PROCEDURE; исполняемая процедура также может возвращать какие-либо значение вызвавшему приложению.

Оба типа хранимых процедур создаются одной и той же командой CREATE PROCEDURE и имеют фактически одинаковый синтаксис. Различие состоит в особенностях написания процедуры и в том, как ее планируется использовать.

Фактически одну и ту же процедуру можно использовать и в качестве исполняемой и возвращающей набор записей, но в основном (и я настойчиво это рекомендую) процедура разрабатывается либо для использования в качестве источника данных либо для непосредственного вызова.

Права доступа

Для использования хранимой процедуры пользователь должен быть либо создателем этой процедуры, либо у него должно быть право EXECUTE (выполнение) для нее.

Хранимые процедуры в свою очередь иногда нуждаются в правах доступа к таблицам или отображениям, непосредственно к кото

рым пользователь не имеет прав доступа. Более подробно о распределении прав доступа см. § 1.3.15.

Создание хранимой процедуры

Хранимая процедура создается с использованием SQL предложения CREATE PROCEDURE, синтаксис которого будет рассмотрен ниже. Хранимая процедура состоит из заголовка и тела процедуры.

Заголовок включает в себя:

- имя процедуры, которое должно быть уникальным в пределах пространства имен процедур, таблиц и отображений базы данных;
- необязательный список параметров процедуры, которые передаются ей вызывающим приложением;
- необязательным списком возвращаемых значений.

Тело процедуры состоит из следующих разделов:

- необязательного списка локальных переменных;
- блока операторов процедурного языка InterBase, начало и окончание которого определяется ключевыми словами BEGIN и END. Такой блок (составной оператор) может включать в себя другие блоки, допуская неограниченную их вложенность.

Внимание! Так как каждое предложение в теле хранимой процедуры должно быть отделено точкой с запятой, необходимо описать другой терминальный символ для всей процедуры в целом. В ISQL используйте команду SET TERM для замены терминального символа, а после завершения тела триггера другую команду SET TERM, которая определяет терминальный символ обратно как точка с запятой. В некоторых средствах визуальной разработки баз данных (например, QuickDesk, см. § 1.3.15) терминаторы заменяются автоматически, поэтому нет необходимости их выполнять вышеописанные действия.

Синтаксис команды создания хранимой процедуры выглядит следующим образом:

```
CREATE PROCEDURE имя_процедуры
[(параметр тип [,параметр тип …])]
[RETURNS (параметр тип [,параметр тип …])]
AS
<тело_процедуры>;

<тело_процедуры>=
[<список_переменных>]
<блок>
```

< список_переменных >=
    DECLARE VARIABLE переменная тип;
    [DECLARE VARIABLE переменная тип; …]

<блок> =
BEGIN
    <составной_оператор>
    [<составной_оператор>…]
END

<составной_оператор>={<блок> | оператор;}

Описание аргументов команды:

Таблица 8

Аргумент	**Описание**
имя_процедуры	Должно быть уникальным в пространстве имен всех процедур, таблиц и отображений базы данных.
параметр тип	Входной параметр, используемый вызывающей программой для передачи значения процедуре; *параметр* – имя параметра, должно быть уникально среди имен всех переменных данной процедуры; *тип* – тип данных InterBase.
RETURNS параметр тип	Возвращаемый параметр, используемый для передаче значения вызывающей программе; параметр и тип соответствуют предыдущему описанию; Процедура возвращает значения по достижению конца тела процедуры, оператора EXIT или SUSPEND.
AS	Ключевой слово, разделяемое заголовок и тело процедуры.
DECLARE VARIABLE переменная тип;	Объявление локальной переменной, используемой только в этой процедуре; каждое описание должно начинаться оператором DECLARE VARIABLE и заканчивать точкой с запятой. Имя локальной переменной должно быть уникально для данной процедуры.
оператор	Любой одиночный оператор процедурного языка InterBase. Любой оператор, за исключением составного, должен заканчиваться

Аргумент	Описание
	точкой с запятой.

Ошибки при создании хранимой процедуры. InterBase может генерировать ошибки при разборе процедуры в случае некорректного использование синтаксиса. Сообщение об ошибке выглядит следующим образом:

```
Statement failed, SQLCODE = -104
Dynamic SQL Error
-SQL error code = -104
-Token unknown - line 4, char 9
-tmp
```

С помощью ключевых слов line и char указывается строка и столбец, возле которого расположен ошибочный оператор. Нумерация линий начинается с команды CREATE PROCEDURE, а не от начала файла описания данных. Так же указывается вид ошибки, в данном случае это Token unknown – неизвестная лексема. Ошибочный оператор может быть расположен непосредственно в указанном месте или до этого места. В случае сомнений необходимо проверить всю строчку для определения причины ошибки.

Процедурный язык InterBase.

Процедурный язык InterBase является готовым языком программирования для хранимых процедур и триггеров. Он включает в себя:

команды SQL для манипулирования данными: INSERT, UPDATE, DELETE, а также SELECT, возвращающий одну строку; также допустимо использование курсоров;

- операторы и выражения SQL, в том числе пользовательские функции, установленные на сервере и генераторы;
- расширения языка SQL, включая оператор присваивания, операторы управления, контекстные переменные, операторы генерации сообщений, исключения и операторы обработки ошибок и исключений.

Хотя хранимые процедуры и триггеры используются совершенно разным образом и для совершенно разных целей, и те и другие используются процедурным языком InterBase. И триггеры и процедуры могут вызывать любые операторы процедурного языка, за некоторыми исключениями:

- контекстные переменные доступны только для триггеров;
- передаваемые и возвращаемые параметры, а также операторы EXIT и SUSPEND доступны только в процедурах.

Следующая таблица описывает все операторы процедурного языка в общем.

Таблица 9

Оператор	**Описание**
BEGIN … END	Описывает составной оператор (блок), который выполняется как один; ключевое слово BEGIN начинает составной оператор, а ключевое слово END заканчивает его. Составной оператор **никогда** не должен заканчиваться точкой запятой.
переменная = выражение	Оператор присвоения, которые присваивает значения *выражения переменной* (локальная переменная, входный и выходной параметр).
/* текст */	Комментарии. Могут располагаться на нескольких строчках.
EXCEPTION имя_исключения	Вызывает указанное исключение. Исключение – описанная пользователем ошибка; может быть обработана оператором WHEN.
EXECUTE PROCEDURE имя_процедуры [переменная [,переменная…]] RETURNING_ЗНАЧUES [переменная [, переменная…]]	Вызывает хранимую процедуру *имя_процедуры*, с необязательными входными параметрами непосредственно за именем, необязательные возвращаемые параметры описываются после ключевого слова RETURNING_ЗНАЧUES Допускается вложенность процедур и рекурсия. Входные и возвращаемые параметры должны быть переменными, описанными в процедуре.
EXIT	Переход на завершающий процедуру END.
FOR команда_select DO оператор	Повторяет составной оператор, описанный после ключевого слова DO для каждой записи, возвращаемой запросом *команда_select* *команда_select* – обычный запрос SQL, за исключение того, что в его окончании должен быть добавлен оператор INTO.
оператор	Простой или составной оператор процедурного языка.
IF (условие) THEN оператор	Проверяется *условие*, и если оно истинно, тогда выполняется оператор, следующий за

Оператор	Описание
ELSE оператор	ключевым словом THEN, иначе выполняется оператор после слова ELSE.
POST_EVENT имя_сообщения	Генерирует сообщение *имя_сообщения*.
SUSPEND	Для процедуры, возвращающей набор данных: – приостанавливает выполнение хранимой процедуры до тех пор, пока вызывающее приложение не затребует следующую запись; – возвращает выходные параметры (ели таковые имеются) вызывающему приложению. Не рекомендуется использовать в исполнимых процедурах.
WHILE (условие) DO оператор.	Оператор цикла. Пока условие истинно выполняется оператор, указанный после ключевого слова DO.
WHEN {ошибка [, ошибка] | ANY} DO оператор	Оператор обработки ошибок. При возникновении одной из перечисленных *ошибок* выполняется *оператор*. В случае наличия оператора WHEN он должен находится в самом конце блока перед завершающим оператором END; *ошибка*: *EXCEPTION имя_исключения*, *SQLCODE код_ошибки* или *GDSCODE номер*. ANY: обработка любой ошибки.

§ 1.3.12 Исключительные ситуации

Исключительная ситуация – это поименованное сообщение об ошибке, которое можно сгенерировать в хранимой процедуре или триггере. При возникновении исключительной ситуации связанное с ней сообщение возвращается клиентской программе и приводит к прекращению выполнения процедуры или триггера, вызвавших эту ситуацию, за исключением случаев обработки ошибок с помощью конструкции WHEN.

Так же, как и хранимые процедуры, исключительные ситуации описываются и сохраняются в базе данных, после чего они могут использоваться процедурами и триггерами. Перед использованием соз

данные исключительные ситуации должны быть подтверждены в контексте соответствующей транзакции.

Для создания исключительной ситуации используется следующая команда:

CREATE EXCEPTION name ‘<сообщение>’;

Пример:

CREATE EXCEPTION acc_overdraw ‘Превышены расходы по счету’.

Для изменения сообщения, связанного с исключительной ситуацией, следует использовать команду

ALTER EXCEPTION name ‘<сообщение>’;

Возможностью изменения сообщения обладает только создатель исключительной ситуации. Сообщение можно менять даже в том случае, если оно используется объектами базы данных. Если сообщение используется хранимой процедурой или триггером, для его удаления Вам необходимо изменить или удалить соответствующую процедуру или триггер.

Для удаления исключительной ситуации воспользуйтесь командой

DROP EXCEPTION name;

Имеются следующие ограничения на удаление исключительных ситуаций:

- только создатель исключительной ситуации может удалить ее;
- не могут быть удалены исключительные ситуации, задействованные в триггерах и хранимых процедурах;
- не могут быть удалены исключительные ситуации, используемые в настоящее время (до подтверждения или отмены транзакции, в контексте которой они возникли)

Генерация исключительной ситуации. Для активации исключительной ситуации в хранимой процедуре или триггере необходимо выполнить следующую команду:

EXCEPTION имя;

При возникновении исключительной ситуации происходят следующие действия:

- прекращается выполнение процедуры или триггера, в котором возникла исключительная ситуация и отменяются все действия, выполненные (прямо или косвенно, например, через другие триггеры) этой процедурой;
- возвращает сообщение об ошибке, связанное с этой исключительной ситуацией, клиентскому приложению.

§ 1.3.13 Обработка ошибок

Процедуры и триггеры могут обрабатывать три вида ошибок:

- исключительные ситуации, вызванные командой EXCEPTION в выполняемой в данный момент процедурой, во вложенной процедуре или в триггере, вызванным действиями такой процедуры;
- ошибки SQL, описываемые кодом ошибки SQLCODE;
- ошибки сервера Interbase, описываемые кодом ошибки GDSCODE;

Предложение WHEN ANY обрабатывает все три вида ошибок.

Подробную информацию об кодах ошибок SQL и Interbase смотрите в *Language Reference*.

Синтаксис предложения WHEN имеет следующую структуру:

```
WHEN {<ошибка> [, <ошибка>] | ANY}
DO оператор
<ошибка> = {EXCEPTION имя| SQLCODE номер |
GDSCODE код}
```

Если в процедуре или триггере используется конструкция WHEN, она должна быть последней в блоке BEGIN…END. Так же она должна следовать после команды SUSPEND, если таковая имеется.

Обработка исключительных ситуаций. В случае возникновения исключительной ситуации имеется возможность не прервать выполнение процедуры или триггера, а попытаться предпринять какие-либо действия для исправления ситуации. При возникновении исключительной ситуации происходит следующая последовательность действий:

- поиск предложения WHEN, которое обрабатывает данную исключительную ситуацию; если же его нет, прерывается выполнения текущего блока BEGIN…END, содержащего вызов исключительной ситуации и отмена всех действий, выполненных в этом блоке;
- повторение предыдущего пункта для всех родительских блоков BEGIN…END; если во всей процедуре или триггере нет предложения WHEN, то процедура или триггер завершается и производится отмена всех действий, выполненных процедурой или триггером;
- если предложение соответствующее WHEN найдено, выполняется оператор обработки ошибки; после этого выполнения процедуры продолжается с оператора, следующего за предложением WHEN;

– Обработанная исключительная ситуация **не возвращает** сообщение об ошибке

Обработка ошибок SQL. Процедуры могут также обрабатывать коды ошибок, возвращаемых в служебной переменной SQLCODE. После выполнения любой команды SQL, SQLCODE содержит код статуса, определяющий успешность/неуспешность этой команды. В SQLCODE помимо этого может содержаться предупреждение, например, о том, что больше не осталось записей для обработке в цикле FOR SELECT.

Например, если в ходе выполнения процедуры осуществляется попытка вставить запись с повторяющимся значением в поле первичного ключа, Interbase возвращает код ошибки -803. В качестве примера обработки такой ошибки можно использовать следующую хранимую процедуру:

```
SET TERM ^ ;

CREATE PROCEDURE NUMBERPROC (A INTEGER, B INTEGER)
RETURNS (e CHAR(60)) AS
BEGIN
    INSERT INTO TABLE1 VALUES (:A, :B);
    WHEN SQLCODE -803 DO
        E = 'Error Attempting to Insert in TABLE1 - Duplicate Value.';
END^

SET TERM ; ^
```

Таким же образом можно обрабатывать и ошибки сервера Interbase. Список ошибок сервера см. в Language Reference.

§ 1.3.14 Пользовательские функции (UDF)

Пользователь может описать свои функции, которые позволяют ему производить нестандартную обработку данных. На платформе Windows функции обычно описаны в виде динамических библиотек (файлы с расширением dll). Для того, что бы использовать такие пользовательские функции, необходимо их объявить в базе данных.

Синтаксис объявления выглядит следующим образом:

```
DECLARE EXTERNAL FUNCTION имя [ тип | CSTRING (число)
[, тип | CSTRING (число) …]]
RETURNS { тип [BY ЗНАЧUE] | CSTRING (число)} [FREE_IT]
ENTRY_POINT 'имя_функции'
```

MODULE_NAME 'имя_модуля';

Ниже описаны основные параметры данной команды:

Таблица 10

Параметр	Описание
имя	Под каким именем будет зарегистрирована данная функция в базе данных.
тип	Тип параметра функции: – все входные параметры передаются в UDF по ссылке; – возвращаемые значения могут передаваться как по ссылке, так и по значению – параметры функции не могут быть массивом
CSTRING (*число*)	Сообщает, что UDF принимает параметр в виде нуль-терминированной строки указанной аргументом *число* длины.
RETURNS	Определяет возвращаемые функцией значения.
BY VALUE	Определяет, что значения возвращаются по значению, а не по ссылке.
FREE_IT	Определяет, что память, занятая под возвращаемое значение, должна быть очищена после завершения работы функции. Используется в том случае, если память под возвращаемое значение была выделена в UDF динамически.
имя_функции	Имя функции, как оно описано в исходной библиотеке.
имя_модуля	Имя библиотеки, в которой находится нужная функция; месторасположение библиотеки в файловой системе может определяться следующими критериями: – библиотека находится в подкаталоге UDF того каталога, где проинсталлирован InterBase сервер; – библиотека находится в каком-либо другом каталоге; в этом случае кроме ее имени необходимо указать также полный путь до нее.

Ниже приведен список предописанных UDF, поставляемых с InterBase. Библиотека с этими функция находится в подкаталоге *UDF* каталога, в который проинсталлирован сервер и называется *ib_udf.dll*.

Команды описания для подключения этих функций находятся в *examples\UDF\ib_udf.sql*.

Функция	**Описание**	**Входные параметры**	**Возвращаемые значения**
abs	Вычисление абсолютного значения числа	double precision	double precision
acos	Вычисление значения функции арккосинуса, если аргумент выходит за границы [-1, 1], то возвращается NaN	double precision	double precision
ascii_char	Возвращает символ, соответствующий числовому коду	integer	char(1)
ascii_val	Возвращает числовой код, соответствующий данному символу	char(1)	integer
asin	Вычисление значения функции арксинуса, если аргумент выходит за границы [-1, 1], то возвращается NaN	double precision	double precision
atan	Вычисление значения функции арктангенса	double precision	double precision
atan2	Возвращает значения арктангенса для выражения параметр 1/параметр 2	double precision	double precision
bin_and	Двоичная операция “И” над двумя целыми числами	integer	integer
bin_or	Двоичная операция “ИЛИ” над двумя целыми числами	integer	integer
bin_xor	Двоичная операция “Исключающее ИЛИ” над двумя целыми числами	integer	integer
ceiling	Выполняет округление сверху (находит число, которое больше или равно данному)	double precision	double precision
cos	Возвращает косинус числа. Если исходное число по абсолютной величине превышает 263, то происходит потеря точности, в результате чего возникает соответствующая	double precision	double precision

Функция	Описание	Входные параметры	Возвращаемые значения
	ошибка и возвращается неопределенное значение		
cosh	Возвращает гиперболический косинус числа. См. так же примечание к функции косинус.	double precision	double precision
cot	Возвращает котангенс числа.	double precision	double precision
div	Выполняет целочисленное деление	integer	integer
floor	Выполняет округление вниз.	double precision	double precision
ln	Вычисляет натуральный логарифм числа	double precision	double precision
log	Функция log(x, y) вычисляет логарифм числа y по основанию x	double precision	double precision
log10	Вычисляет десятичный логарифм числа	double precision	double precision
lower	Возвращает исходную строку, приведенную к нижнему регистру. Эта функция может не работать с международными символов и символами, соответствующими стандарту ascii. Обрабатывает строки длиной до 32767 символов	cstring(80)	cstring(80)
ltrim	Удаляет пробелы в начале строки. Обрабатывает строки длиной до 32767 символов	cstring(80)	cstring(80)
mod	Возвращает остаток от целочисленного деления	integer	double precision
pi	Возвращает число π, равное 3.1415…	–	double precision
rand	Возвращает псевдослучайное число от 0 до 1. Генератор псевдослучайных чисел инициализируется текущим вре	–	double precision

Функция	Описание	Входные параметры	Возвращаемые значения
	менем.		
rtrim	Удаляет завершающие пробелы в конце строки. Обрабатывает строки длиной до 32767 символов	cstring(80)	cstring(80)
sign	Возвращает соответственно -1, 0 или 1 для аргумента, имеющего отрицательное, нулевое или положительное значение	double precision	integer
sin	Возвращает синус числа. Если исходное число по абсолютной величине превышает 263, то происходит потеря точности, в результате чего возникает соответствующая ошибка и возвращается неопределенное значение	double precision	double precision
sinh	Возвращает гиперболический синус числа. Если исходное число по абсолютной величине превышает 263, то происходит потеря точности, в результате чего возникает соответствующая ошибка и возвращается неопределенное значение	double precision	double precision
sqrt	Возвращает квадратный корень числа	double precision	double precision
substr	Функция substr(s,m,n) возвращает подстроку строки s начиная с символа с позицией m по символ с позицией n. Обрабатывает строки длиной до 32767 символов	cstring(80), smallint	cstring(80)
strlen	Возвращает длину строки	cstring(80)	integer
tan	Возвращает тангенс числа. Если исходное число по абсолютной величине превышает 263, то происходит потеря	double precision	double precision

Функция	Описание	Входные параметры	Возвращаемые значения
	точности, в результате чего возникает соответствующая ошибка и возвращается неопределенное значение		
tanh	Возвращает гиперболический тангенс числа. Если исходное число по абсолютной величине превышает 263, то происходит потеря точности, в результате чего возникает соответствующая ошибка и возвращается неопределенное значение	double precision	double precision

Пример описания функции *sqrt*:

```
DECLARE EXTERNAL FUNCTION sqrt
        DOUBLE PRECISION
        RETURNS DOUBLE PRECISION BY ЗНАЧUE
        ENTRY_POINT 'IB_UDF_sqrt' MODULE_NAME 'ib_udf';
```

§ 1.3.15 Определение прав доступа к БД

Права доступа Interbase управляют привилегиями доступа на табличном уровне. Привилегиями доступа называется список операций, разрешенных для выполнения над объектами базы данных определенному пользователю. Команда GRANT назначает привилегии доступа к объекту базы данных указанному пользователю, роли или другому объекту базы данных, а также позволяет указывать принадлежность пользователя какой-либо группе. Для удаления привилегий доступа необходимо использовать команду REVOKE.

Для создания пользователей необходимо воспользоваться программой IBConsole, либо каким-либо визуальным средством разработки баз данных Interbase других производителей, так как в этой реализацией SQL Interbase нет специальных команд создания пользователей. Список пользователей обычно хранится в специальной базе данных isc4.gdb, однако на Unix-системах также может браться из /etc/password как на сервере, так и на клиентской машине.

В InterBase определены следующие привилегии доступа к различным объектам:

Таблица 11

Права	Описание

Права	Описание
ALL	Даются все права доступа для таблицы: SELECT, INSERT, UPDATE, DELETE и REFERENCES.
SELECT	Чтение данных из объекта
INSERT	Добавление записей
UPDATE	Изменение существующих записей (в том числе с указанием полей, которые разрешено менять)
DELETE	Удаление записи
REFERENCES	Использование для связывания с другими таблицами на стороне один (в том числе с указанием разрешенных полей).
EXECUTE	Выполнение хранимой процедуры.

Команда GRANT может использоваться следующими способами:

- Разрешать SELECT, INSERT, UPDATE, DELETE и REFERENCES права доступа к таблице пользователям, ролям, триггерам, хранимым процедурам и отображениям (view) с указанием необязательного WITH GRANT OPTION (см. ниже).
- Разрешать SELECT, INSERT, UPDATE и DELETE права доступа к отображению пользователям, ролям, триггерам, хранимым процедурам и отображениям (view) с указанием необязательного WITH GRANT OPTION (см. ниже).
- Присваивать роли пользователей (с необязательным параметром WITH ADMIN OPTION).
- Разрешать право доступа EXECUTE к хранимой процедуре пользователям, ролям, триггерам, другим хранимым процедурам и отображениям (view) с указанием необязательного WITH GRANT OPTION (см. ниже).

Примечания: ALL не дает право доступа EXECUTE. Права SELECT, INSERT, UPDATE, DELETE и REFERENCES могут произвольно комбинироваться в одной команде GRANT или REVOKE. Команды, разрешающие или запрещающие право EXECUTE или определяющие список пользователей роли, не могут включать в себя другие права.

Ограничения доступа по умолчанию. Все объекты базы получают защиту от неавторизованного доступа при их создании. Только создатель объекта, так же являющийся его владельцем, имеет все права доступа к этому объекту и только он может разрешить те или иные права доступа другим пользователям.

На сервере Interbase также всегда имеется пользователь SYSDBA, имеющий полный доступ к любому объекту любой базы, так называемый суперпользователь. На некоторых операционных платформах, поддерживающих концепцию суперпользователя, администратора или пользователя с правами root, такие пользователи также имеют полный доступ к любым объектам БД.

Следует заметить, что для несанкционированного доступа без ограничений к какой-либо базе достаточно скопировать ее на машину с установленным сервером Interbase и известным паролем пользователя SYSDBA. Поэтому администратору сервера необходимо заботится о физической недоступности для потенциальных злоумышленников файла или файлов, содержащих базу данных.

Роли. Interbase 6.x поддерживает возможность назначения SQL привилегий группам пользователей, что осуществляется с помощью так называемых ролей. Для использования роли необходимо выполнить четыре действия.

1. Создать роли, используя команду CREATE ROLE.
2. Назначить права доступа роли с помощью команды GRANT.
3. Включить пользователей в роль с помощью команды GRANT.
4. Указать роль при подключении к базе данных.

Назначения прав доступа к таблице или отображению.

Команда присвоения прав доступа к таблице или отображению имеет следующий синтаксис:

```
GRANT <привилегии> ON [TABLE] {имя_таблицы |
имя_отображения}
TO { <объект> | <список_пользователей> | GROUP UNIX_group}

<привилегии> = {ALL [PRIVILEGES] | <список_привилегий>}

<список_привилегий > = SELECT
| DELETE
| INSERT
| UPDATE [(поле [, поле …])]
| REFERENCES [( поле [, поле …])]
[, <список_привилегий> …]

<объект> = PROCEDURE имя_процедуры
| TRIGGER имя_триггера
| VIEW имя_отображения
| PUBLIC
[, <объект> …]
```

<список_пользователей> = [USER] имя_пользователя
| имя_роли
| Unix_user}
[, <список_пользователей> …]
[WITH GRANT OPTION]

<роли> = имя_роли [,имя_роли …]

Команда GRANT позволяет разрешать доступ к таблице или отображению списку пользователей или других объектов базы данных. Разрешение доступа для других объектов базы данных может быть необходимо для обеспечения необходимой степени безопасности. Например, пользователь *op1* не имеет каких-либо прав доступа к таблице *t1*, однако, разработчик решил разрешить этому пользователю доступ SELECT к отображение *view_t1*, основанному на таблице *t1*. Для реализации такого правила секретности мы должны дать право SELECT пользователю *op1* на отображение *view_t1*. Но так как пользователь не имеет прав доступа к *t1*, то при попытке открыть *view_t1* будет выдано сообщение о запрете доступа. Для того, что бы решить эту проблему, необходимо дать право доступа SELECT отображение *view_t1* на таблицу *t1*:

GRANT SELECT ON view_t1 TO op1;

GRANT SELECT ON t1 TO VIEW view_t1;

Можно производить делегирование нескольких прав одновременно нескольким объектам, пользователям или ролям. Например:

GRANT SELECT, INSERT ON t1 TO op1, op2, sysoperator;

GRANT SELECT, INSERT, UPDATE ON t1 TO VIEW view_t1,
PROCEDURE execute_me;

Также имеется возможность указать права доступа не к таблице целиком, а к отдельным полям (только для прав UPDATE и REFERENCES):

GRANT SELECT, INSERT, UPDATE(name, dob), REFERENCES(id)
ON t1 TO smith;

Используя конструкцию WITH GRANT OPTION можно разрешить делегирование соответствующих полномочий пользователю, например, после выполнения команды

GRANT SELECT, INSERT ON t1 TO op1 WITH GRANT OPTION,
op2, sysoperator WITH GRANT OPTION;

пользователи *op1* и *sysoperator* могут делегировать права SELECT и INSERT к таблице *t1* другим пользователям (см. так же ниже).

С помощью ключевого слова ALL можно дать **полный доступ** к таблице или отображению, который включает в себе права SELECT, INSERT, UPDATE, DELETE и REFERENCES (только для таблиц). Назначение доступа к какому-либо объекту встроенной группе PUBLIC дает **доступ** к этому объекту **любому пользователю** (в группу PUBLIC входят все пользователи сервера Interbase).

Назначение прав доступа к отображению(view).

Для пользователя отображения выглядят как обычные таблицы, однако на самом деле между таблицами и отображениям есть существенное отличие – содержимое отображений не хранится в базе, хранится только описание отображений в виде запроса к базовым таблицам. Поэтому любые действия с отображением – это действия с базовыми таблицами.

Синтаксис для назначения прав доступа к отображению имеет тот же вид, что и к таблице, за исключением того, что для отображения нельзя использовать право REFERENCES.

Иногда бывает необходимо ограничить пользователю права доступа к базовым таблицам, предоставив при этом доступ к отображению, базирующемуся на этих таблицах. В этом случае отображению нужно назначить соответствующие права доступа к базовым таблицам (пример см. выше).

Назначение прав доступа к хранимым процедурам.

Единственным правом доступа к хранимым процедурам является право EXECUTE, позволяющее указанным пользователям запускать эту хранимую процедуру. Синтаксис команды GRANT в этом случае будет следующим:

```
GRANT EXECUTE ON PROCEDURE имя_процедуры
TO { <объект> | <список_пользователей>}
```

Права доступа к процедурам могут иметь другие объекты базы данных, в частности, процедуры и триггеры, которые вызывают другие процедуры (при условии, что пользователь, от имени которого выполняются такие процедуры или триггеры, не имеет доступа к вызываемым процедурам).

Примеры:

```
GRANT EXECUTE ON PROCEDURE p1 TO op1, op2;

GRANT EXECUTE ON PROCEDURE p1 TO
    PROCEDURE p2, p3;

GRANT EXECUTE ON PROCEDURE p1 TO TRIGGER tr_t2_1;
```

В первой команде право EXECUTE дается пользователям *op1* и *op2*, вторая команда разрешает вызывать процедуру *p1* из процедур *p2*

и *p3* (заметьте, что если доступ разрешается нескольким процедурам, то ключевое слово PROCEDURE указывается один раз перед списком процедур, то же самое верно и для триггеров), последняя команда разрешает вызов процедуры *p1* из триггера *tr_t2_1*;

Внимание! Если командой GRANT предоставляется право EXECUTE встроенной группе PUBLIC, то нельзя в этой же команде давать доступ другим пользователям или объектам БД.

Создание и использование ролей. Как было описано выше, для работы с ролью нужно выполнить четыре действия, каждое из которых подробно описано ниже.

1. Создать роли, используя команду CREATE ROLE:

   ```
   CREATE ROLE имя_роли;
   ```

2. Назначить права доступа роли с помощью команды GRANT (подробнее см. выше):

   ```
   GRANT привилегии ON объект TO имя_роли;
   ```

3. Включить пользователей в роль с помощью команды GRANT:

   ```
   GRANT { имя_роли [,имя_роли …]} TO
   {PUBLIC | {[USER] имя_пользователя [, [USER]
   имя_пользователя …]} }
   [WITH ADMIN OPTION];
   ```

   Здесь PUBLIC, как уже было отмечено выше, является встроенной группой в которую входят все пользователи сервера. Конструкция WITH ADMIN OPTION позволяет предоставить право перечисленным пользователям добавлять в указанные роли других пользователей (подробней см. ниже).

4. Указать роль при подключении к базе данных.

   Обычно клиентские программы позволяют указывать кроме имени входа пользователя и его пароля также и имя роли. При разработки своих собственных клиентских программ также необходимо предоставлять эту возможность. Если пользователь указывает при соединении с сервером и роль, то сервер, кроме обычной процедуры аутентификации, проверит, входит ли пользователь в указанную роль и только после этого разрешит работу с базой данных.