Алексей Ковязин, Сергей Востриков

Мир InterBase

Архитектура, администрирование и разработка приложений баз данных в InterBase/Firebird/Yaffil

КУДИЦ-ОБРАЗ
Москва • 2003

ББК 32.973-018
Ковязин А., Востриков С.
Мир InterBase. Архитектура, администрирование и разработка приложений баз данных в InterBase/Firebird/Yaffil. Издание 2-е, дополненное –
М.: КУДИЦ-ОБРАЗ, 2002. – 496 с.

ISBN 5-93378-074-X

Книга представляет собой первое в России издание, посвященное использованию популярного сервера баз данных InterBase и написанию клиентских приложений для него. Книга предназначена как для начинающих разработчиков, так и для практикующих опытных программистов приложений для InterBase.

На начинающих разработчиков ориентирована первая часть – «Быстрый старт», содержащая описание ключевых понятий и примеров работы с InterBase.

Во второй части книги приведены материалы по разработке клиентских приложений на основе самых современных и эффективных средств доступа к InterBase – FIBPlus, OLE DB IBProvider, а также примеры использования InterBase API. Рассмотрены вопросы применения драйверов ODBC и JDBC для InterBase.

В третьей части книги подробно описаны вопросы администрирования InterBase – починки баз данных, оптимизации работы сервера и многое другое. Также в этой части приведено описание архитектуры InterBase, затрагивающее широкий круг вопросов.

В приложениях к книге приведены переводы необходимых каждому разработчику InterBase документов, представлен обзор российского клона InterBase 6.x Yaffil, а также помещен глоссарий и список ключевых слов InterBase.

Ковязин А. Н., Востриков С. М.
Мир InterBase. Архитектура, администрирование и разработка приложений баз данных в InterBase/Firebird/Yaffil

Учебно-справочное издание

Корректор М. Матёкин
Макет А. Ширкевич

Лицензия ЛР № 071806 от 2.03.99. НОУ "ОЦ КУДИЦ-ОБРАЗ"
119034, Москва, Гагаринский пер., д. 21, стр. 1. Тел.: 333-82-11, ok@kudits.ru

Подписано в печать 28.03.2003.
Формат 70x100/16. Бум. офсетная. Печать офс.
Усл. печ. л. 40. Тираж 3000. Заказ

ISBN 5-93378-074-X

Предисловие к 3-му изданию

Прошло 3 года с момента публикации 1-го издания «Мир InterBase». За это время изменилось многое. Из основных событий можно отметить выход новых версий серверов Borland InterBase и Firebird, слияние проекта Yaffill с Firebird в версии 2.0, а также публикацию первой англоязычной книги по Firebird, написанной Хелен Борри - замечательной австралийской энтузиасткой этого проекта - “Firebird SQL Development Guide”.

Сейчас выходит 3-е издание «Мира InterBase». Прежде всего, это говорит о росте сообщества InterBase и Firebird как количественном – суммарный тираж только первых двух изданий составил 8 тысяч экземпляров, так и качественном - число российских проектов на базе InterBase и Firebird измеряется десятками тысяч. Теперь гораздо реже можно услышать пренебрежительное (и чрезвычайно неграмотное) мнение об InterBase как о «компоненте Delphi».. Постоянный спрос на специалистов по InterBase/Firebird существует и на рынке труда, что подтвержает нашу глубокую уверенность в нужности этой книги.

3-е издание этой книги во многом повторяет первые два, так как изложенные принципы работы с семейством серверов InterBase и Firebird остались неизменными. Конечно, было внесено множество исправлений, дополнений и пояснений, часть устаревших сведений, а также несбывшиеся прогнозы были удалены.

Предисловие к 2-му изданию

Первое издание этой книги вышло 4 сентября 2002 года тиражом в 3000 экземпляров. Ровно через 3 месяца тираж бы распродан.

Успех этой книги (а для компьютерной книги, посвященной СУБД, это действительно успех) обозначил потребности русскоязычных пользователей СУБД InterBase и Firebird, а также подтвердил слух о том, что InterBase – «серый кардинал» на рынке СУБД в России. Первое издание было посвящено сразу 3 серверам СУБД – InterBase, Firebird и Yaffil. Однако за прошедшее с момента окончания работы над книгой время многое изменилось, и поэтому нам, авторам книги, пришлось значительно доработать ее, в связи с чем и появилось второе издание.

В процессе работы над 2-м изданием было исправлено несколько сотен опечаток, внесена масса уточнений и дополнений. Очень сильно помогли читатели книги, которые присылали нам найденные ошибки и опечатки, а также задавали уточняющие вопросы.

Помимо исправлений и уточнений, мы дополнили книгу главами по библиотеке InterBase Express (IBX), а также несколькими главами, посвященными последним версиям СУБД InterBase, Firebird и Yaffil.

Фактически эта книга является последней книгой, посвященной сразу 3 серверам – Borland InterBase, Firebird и Yaffil. Почему, спросит уважаемый читатель?

29 ноября 2002 года было объявлено о выходе новой версии СУБД InterBase 7. Событие само по себе радостное, и в соответствующей главе данной книги мы рассмотрим, какие технические нововведения принесла нам семерка, а сейчас остановимся на «политической» подоплеке выпуска InterBase 7.

Прежде всего, можно сказать, что выпуск 7-й версии ознаменовал собой окончание неразберихи с версиями и клонами InterBase. Если внимательно проследить предшествующую историю, то обнаружим, что к 2000 году сложилась довольно странная ситуация – существовало сразу несколько версий вроде бы одного и того же сервера, но от разных производителей. Речь идет о том, что серверы баз данных InterBase 6.0.x, Firebird 1.0 и даже InterBase 6.5 были практически полностью совместимы друг с другом – базу данных от любого из этих серверов можно было использовать с любым другим даже без процесса backup/restore (хотя это и не рекомендуется делать).

«Похожесть» этих серверов баз данных постепенно стала перерастать в большую проблему. Прежде всего – для компании Borland, которая продавала сертифицированные версии сначала InterBase 6, а затем выпустила полностью платный InterBase 6.5. Продвинутые пользователи знали, что абсолютно бесплатный Firebird 1.0 предоставляет им те же возможности, что и 6.0, и это не очень хорошо влияло на продажи Borland InterBase 6.0/6.5.

В то же время все пользователи по привычке называли Firebird «InterBase», а некоторые российские пользователи вообще именовали эту СУБД не иначе как «халявный интербейз». Разумеется, это не устраивало разработчиков Firebird, которые уже вложили в исходные коды 6-го InterBase Open Source массу сил и своего труда. Получалось, что лавры их трудов так или иначе переходили к Borland.

Короче говоря, назрела необходимость четкого позиционирования серверов InterBase, Firebird и чуть позже российского Yaffil.

С выходом InterBase 7 позиционирование этих серверов значительно облегчилось. Декларируется, что семерка совместима только с InterBase 6.5, и рекомендуется проводить миграцию только через backup/restore. В сущности, никто не мешает перевести базу под семерку и с Firebird 1.0, и с InterBase 6.0 – путем нескольких последовательных миграций. Более того, в семерке есть поддержка 1-го диалекта для облегчения миграции с еще более старых версий, вроде 5.6. Но, как бы то ни было, решительный настрой компании Borland совершенно очевиден и логичен – Borland InterBase должен быть совместим только с Borland InterBase. Как вы можете узнать из главы, посвященной Firebird 1.5, аналогичной «сепаратистской» позиции придерживаются и Firebird Developers. И это очень хорошо – вместо одного отличного сервера баз данных у нас будут 2 (или даже 3, если считать Yaffil).

Другое дело, что все пользователи InterBase 4.x–5.x–6.x и Firebird 1.0 поставлены перед однозначным выбором – необходимо выбрать СУБД для своего развития и двигаться уже вместе с ней, не оглядываясь на другие варианты.

Выберете коммерческий Borland InterBase или окунетесь в мир Open Source с Firebird – решать вам. Пройдет два года, и разница между функциями этих серверов приведет к необходимости написать несколько разных книг.

Предисловие к 1-му изданию

Слухи о моей смерти сильно преувеличены!
Марк Твен

Мы очень рады предложить вниманию читателя первую книгу в России, которая посвящена серверу баз данных InterBase. Несмотря на широкую известность и распространенность этого замечательного продукта, до сих пор разработчикам было очень трудно найти какие-либо справочные материалы по InterBase, особенно на русском языке.

Сейчас ситуация меняется. Первым шагом было появление русскоязычного сайта по InterBase http://ib.demo.ru (с недавнего времени он доступен по адресу http://www.ibase.ru), созданного усилиями Дмитрия Кузьменко. Вторым крупным событием стала публикация русского издания Media Kit – полной документации для Borland InterBase 5, переведенной на русский язык. Каждый день мы видим, как увеличивается число разработчиков, использующих InterBase в своих приложениях. Теперь настала пора создания книги, которая помогла бы разработчикам приложений баз данных понять InterBase лучше, потому что, несмотря на свою простоту, этот сервер сочетает в себе огромный спектр возможностей.

InterBase является кроссплатформенным продуктом, поддерживающим большое количество различных операционных систем, включая Microsoft Windows NT, Windows 2000, Windows XP, Windows 98/ME, Linux и несколько Unix-платформ. InterBase отличается чрезвычайно низкими системными требованиями и при этом высокой производительностью и легкостью администрирования. Вы можете работать с InterBase, используя несколько сетевых протоколов: TCP/IP, NetBEUI/named pipes, IPX/SPX.

Одной из основных особенностей InterBase, пожалуй, можно считать версионную архитектуру, которая обеспечивает уникальные возможности при многопользовательской работе – пишущие пользователи никогда не блокируют читающих! Помимо этого, версионная архитектура позволяет отказаться от использования протокола транзакций (transaction log), который в других СУБД служит для восстановления базы данных после сбоев, поэтому InterBase обладает очень высокой надежностью и устойчивостью.

Также в InterBase реализован механизм оптимистической блокировки на уровне записи. Это значит, что сервер блокирует только те записи, которые реально были изменены пользователем, и не блокирует всю страницу данных целиком. Эта особенность еще больше снижает вероятность конфликтов при многопользовательском режиме работы.

InterBase полностью совместим со стандартом ANSI SQL 92, а также имеет свое собственное расширение SQL для хранимых процедур и триггеров. В сравнении со многими другими СУБД, InterBase предоставляет очень эффективный механизм триггеров: каждая таблица может иметь большое количество триггеров, которые выполняются автоматически при вставке, изменении или удалении каждой отдельной записи, до или после этих событий. Многие функции существующих СУБД были впервые реализованы в InterBase – это, в частности, обновляемые представления, события (event alerters), многомерные массивы и BLOB-поля. Более того, некоторые механизмы, такие, например, как двухфазное

подтверждение транзакций, до сих пор остаются совершенно уникальными, представленными только в InterBase.

Немаловажной особенностью сервера InterBase является возможность расширения стандартного набора SQL-функций при помощи пользовательских библиотек – User Defined Functions, а также механизмы обработки BLOB-полей на сервере при помощи BLOB-фильтров. Остается только сказать, что InterBase отличается значительной устойчивостью, поскольку специально был спроектирован для применения в Intranet-приложениях, приложениях для мобильных устройств и встроенных приложениях баз данных.

В этой книге мы постарались затронуть самый широкий перечень вопросов, связанных с InterBase: от первоначальной установки до анализа физической структуры файла базы данных, поэтому книга может быть интересна как начинающим разработчикам, так и профессионалам.

Говоря об InterBase, мы прежде всего будем иметь в виду семейство серверов InterBase 6.x, однако большинство включенных в книгу материалов может также относиться и к более ранним версиям InterBase, которые до сих пор используются многими компаниями, т. е. к версиям 5.x и даже 4.x. Но основное внимание, конечно, уделено самым современным версиям InterBase 6.x.

Под семейством серверов InterBase 6.x подразумевается сразу несколько продуктов, поскольку на сегодняшний день существует несколько клонов, основанных на исходном коде Borland InterBase 6.0. Фактически мы включили в книгу описания особенностей Borland InterBase 6.0 Open Edition, Borland InterBase 6.5, Firebird 1.x и Yaffil.

InterBase 6.0 Open Edition, Firebird и Yaffil являются Open Source-продуктами, которые вы можете использовать бесплатно в рамках условий InterBase Public License. Borland InterBase 6.5 доступен в виде 90-дневной TRIAL-версии. Несмотря на такое обилие продуктов, на текущий момент они имеют много общего.

Таким образом, если мы захотим подчеркнуть особенности конкретной версии в том или ином контексте, то мы будем указывать эту версию явным образом. Если же речь идет об InterBase вообще, то это значит, что информация верна для всех существующих (клонов).

Благодарности

Мы никогда не смогли бы написать эту книгу без помощи и участия множества людей. Поэтому мы хотели бы выразить искреннюю признательность всем тем людям, которые помогали создавать эту книгу своими материалами, советами, консультациями и просто поддерживали добрым словом.

Прежде всего хочется поблагодарить издательство «Кудиц-Образ» за то, что оно взяло на себя смелость напечатать первую в России книгу по InterBase.

Особую благодарность мы выражаем Александру Владимировичу Невскому, чье участие и поддержка в создании книги просто неоценимы. Ценные подсказки, подробные консультации и меткие замечания очень помогли нам в работе над книгой. Большое спасибо, Ded!

Хочется искренне поблагодарить людей, предоставивших свои материалы для опубликования: Дмитрия Еманова, Олега иванова, Дмитрия Коваленко, Алексея Карякина и Сергея Мереуцу. Их профессионализм позволил затронуть в книге множество интересных дополнительных тем, материалы по которым обычно недоступны читателю.

Большое спасибо за сообщения об ошибках и опечатках, а также за ценные советы, хочется сказать Бурову Дмитрию, Руслану Бондаренко, Сергею Пронякину, Игорю Лагову, Каратаеву Владимиру, Вячеславу Екимову, Александру Лезликову.

Мир InterBase – это интернациональное сообщество, и наша книга не могла бы существовать в отрыве от наших зарубежных коллег. Мы крайне благодарны госпоже Анне В. Харрисон (Ann W. Harrison) и г-ну Клаудио Р. Вальдераме (Claudio R. Valderrama) за любезное разрешение включить в книгу их материалы, а также за уникальные и исключительно ценные консультации.

Выражаем искреннюю благодарность консультантам книги: Александру Хвастунову и Алексею Флегонтову за то, что они терпеливо читали черновики книги и давали важные и ценные советы.

Особое спасибо мы адресуем Сергею Бузаджи за проверку и рецензирование главы про FIBPlus. Мы искренне благодарны Сергею Уолису и Дмитрию Коваленко за их статьи и примеры по FIBPlus, во многом определившие тематику и уровень изложения материала.

Также хочется сказать спасибо людям, которые поддерживали нас в работе над книгой: Константину Сипачеву, Андрею Борискину, Юрию Котлярову, Наталье Полянской и Олегу Кулькову.

Особая благодарность Дмитрию Кузьменко, создателю и вдохновителю российского InterBase-сообщества.

Авторы

Часть1
Быстрый старт

Установка InterBase

Перед тем как начать разрабатывать приложения баз данных с помощью InterBase, необходимо позаботиться о его установке. Обычно InterBase устанавливают как на сервер, так и на рабочую станцию программиста, разрабатывающего приложение. Разработчику InterBase нужен для внутренних экспериментов и отладки рабочих версий базы данных и программ, а InterBase на сервере используется для тестирования программы пользователями или совместной разработки в случае, если над проектом работает команда. Благодаря своей легковесности и нетребовательности к ресурсам InterBase можно спокойно устанавливать прямо на рабочие станции разработчиков, не беспокоясь о снижении быстродействия. Когда InterBase не обслуживает подключений к базам данных, находясь в ожидании запросов, то занимает памяти меньше, чем такие популярные программы, как ICQ или WinAmp.

Что ставить?

Прежде чем устанавливать InterBase, надо решить, какой из его клонов (версий) мы выберем для работы. Прежде всего надо сказать, какую версию однозначно НЕ РЕКОМЕНДУЕТСЯ использовать. Это бета-версия InterBase 6.0.xx, одна из самых первых версий, выпущенных после объявления InterBase 6 бесплатным. Версия InterBase 6.x послужила основой для выпуска трех клонов – семейства Firebird (версии 1.0 и 1.5), Borland InterBase (6.5, 7.0, 7.1 b 7/5) и Yaffil (1.0).

Сервер Firebird появился вскоре после опубликования исходных кодов InterBase 6.x. Он является абсолютно бесплатным продуктом в открытых кодах (open source). В его создании принимало участие множество бывших сотрудников InterBase Software Corporation.

Самая последняя версия на момент 3-го издания - Firebird 1.5.1 отлично подойдет для освоения материала этой книги и начала работы с InterBase.

Компания Borland продолжает развивать линейку своих коммерческих серверов, последней из которых является 7.1 Service Pack 2. Ее также можно установить и использовать для изучения материала этой книги. Для этих целей можно воспользоваться триал-версией с ограничением времени использования в 90 дней. Эту версию InterBase 7.1 можно скачать с сайта компании Borland.

Помимо Firebird и InterBase, существует еще и российская ветка InterBase 6.x – Yaffil, который отпочковался от Firebird в 2001 году. Yaffil существует только для платформы Windows, и часть его кода оптимизирована именно для этой ОС. В 2004 году разработчики Yaffil влились в команду Firebird и договорились, что эти проекты сольются в версии Firebird 2.0, которая планируется в 2005 году.

Следует отметить, что большинтсво клонов InterBase совместимы между собой и базу данных с одного сервера можно легко перевести под другой с помощью процесса *backup/restore* (см. главу «Миграция» (ч. 4)). Однако стоит помнить о том, что совместимость между Firebird и InterBase становится все меньше, и для переноса базы данных может потребоваться прибегнуть к каскадной миграции.

Многих разработчиков пугает значительное разнообразие версий и потомков InterBase и вызывает чувство неуверенности – много разных СУБД, которые непонятно чем отличаются. На самом деле это совершенно нормальная ситуация, типичная для проекта Open Source и свидетельствующая о том, что и Bolrland InterBase и Firebird активно развиваются. Совет новичкам - просто внимательно следите за новостями (например, на сайте www.interbase-world.com) и будьте в курсе самых последних версий.

Немного «пообщавшись» с InterBase (с любым его клоном), вы легко сможете выбрать, что же больше подходит для конкретной задачи.

Вопрос – где находятся дистрибутивы различных клонов InterBase? Различные варианты дистрибутивов представлены на сайте поддержки данной книги www.interbase-world.com, а также на сайте http://www.ibase.ru. Там же обычно можно получить ссылку на самый «свежий» дистрибутив всех клонов InterBase, а также массу другой полезной информации по использованию InterBase и разработке приложений баз данных.

Непосредственно из первоисточника дистрибутивы и исходные коды InterBase можно взять на сайте http://sourceforge.net/projects/interbase, а дистрибутивы Firebird – на сайте http://sourceforge.net/projects/firebird. В общем, хорошая поисковая машина вам поможет в получении практически любого дистрибутива.

На компакт-диске с Delphi Client/Server и Enterprise Edition всех версий (3.x, 4.x, 5.x и 6.x) также содержится InterBase, обычно версии 5.x/6.x. Его также можно использовать для работы с материалом этой книги, за исключением некоторых моментов, которые описывают функциональность, присущую только некоторым версиям клонов InterBase.

Определившись с версией InterBase, следует выбрать, под какой ОС мы будем работать. Для нашего случая предположим, что работа будет происходить на ОС Windows (версии старше 95 или NT 4.0). Если вы поклонник другой ОС, не расстраивайтесь: существуют дистрибутивы для Linux/Unix, FreeBSD, Solaris, и даже Mac OS X Darwin. Здесь мы рассмотрим процесс установки для платформ Windows и Linux, предполагая, что именно одна из этих ОС установлена на вашем рабочем компьютере.

Если в будущем потребуется «переехать» на другую ОС, нет проблем: это несложная процедура, которую мы рассмотрим в главе «Миграция».

Установка InterBase на платформе Windows

Инсталляторы

Инсталлятор (программа, которая занимается установкой продукта) Borland InterBase 7.x занимает около 70 Мбайт. Размер инсталляции «раздуло» с традиционных 5-10Мбайт из-за того, что теперь установкой занимается универсаль-

ный Java-based инсталлятор InstallAnywere. Реально размер уже установленного InterBase 7.1 по прежнему остается очень небольшим.

Инсталлятор Firebird 1.5.1 по-прежнему компактен, и для каждой платформы существует свой инсталлятор.

Вообще говоря, установка этих серверов ничем кардинально друг от друг от друга не отличается, важно учесть лишь несколько ключевых моментов, связанных именно с установкой IntrerBase/Firebird. Поэтому давайте рассмотрим установку Firebird 1.5.1 (www.firebirdsql.org). Если кто-то очень спешит, то может обратиться к ключевым моментам в конце главы.

Подготовка к установке

Если на компьютере, куда устанавливается Firebird, уже установлена любая версия InterBase, то ее сначала необходимо обязательно остановить. Для этого следует воспользоваться либо «Панелью управления» Windows, апплетом «Службы», если установка идет на Windows NT/2000/XP, либо иконкой InterBase-сервера на панели задач Windows, если устанавливаем на Windows 95/98/Me.

Внимание! Остановка уже существующей версии InterBase при установке новой обязательна. Рекомендуется вообще удалить предыдущую версию, чтобы не сталкиваться с конфликтами обновления файлов. Желательно удалить файлы gds32.dll из всех папок данного компьютера, указанных в переменной пути поиска Path (проще всего это сделать так: поискать на локальных дисках gds32.dll и удалить их отовсюду, кроме папок с дистрибутивами).

Установка

После запуска инсталлятора появится оповещение о том, что ставится именно та версия, которая нам нужна, – в данном случае Firebird 1.5.1. Нажмите Next для перехода к следующему шагу установки. На экране появится текст InterBase Public License. Выберите I agree и перейдите к следующему шагу. Появится окно, в котором предлагается выбрать путь для установки Firebird. Это важная опция. Обычно по умолчанию инсталлятор предлагает путь C:\Program Files\Firebird. Однако часто имеет смысл переопределить этот путь во что-то вроде C:\IBServer.

Выбрав путь для установки, нажмите Next для перехода к следующему шагу установки. При этом на экране появится окно, изображенное на рис. 1.1. (вполне возможно, что дизайн его может отличаться для различных версий Firebird, но смысл остается тем же). На этом шаге мы выберем компоненты, необходимые для работы. Этот выбор достаточно прост – выберите все компоненты, находящиеся в списке. Как видите, они займут весьма скромное место на диске – всего около 16 Мбайт.

Если же на компьютере, куда устанавливается InterBase, существует недостаток дискового пространства, то можете ограничиться только опциями Server for Windows, Client for Windows и Command Line Tools. Абсолютно минимальная установка InterBase описана в главе «Установка InterBase – взгляд изнутри» (ч. 4).

Нажмите Next для перехода к следующему шагу. *Внимание!* В этот момент инсталлятор может выдать сообщение «InterBase is running on this machine…» и прекратить установку. Это означает, что на компьютере уже установлен и за-

пущен InterBase (возможно, он был установлен вместе с Delphi). В этом случае надо удалить предыдущую версию InterBase, согласно рекомендациям раздела «Подготовка к установке». После чего можно продолжить установку.

Если же после выбора компонентов не возникло никаких проблем и началось копирование файлов, то все в порядке, просто дождитесь завершения копирования, затем нажмите Finish.

Вот и все – установка Firebird закончилась. Обратите внимание, что для ее завершения не понадобилась перезагрузка компьютера – сервер уже запустился. Если вы используете ОС Windows 95/98/Me, то работающий InterBase отобразится на панели задач в виде значка среди системных иконок, если NT 4/2000 – то сервер запустится как служба (service) и на экране никак его функционирование не отразится – наблюдать за его статусом и управлять работой можно лишь через апплет «Панели управления» «InterBase Manager» или через аплет «Службы» («Services»).

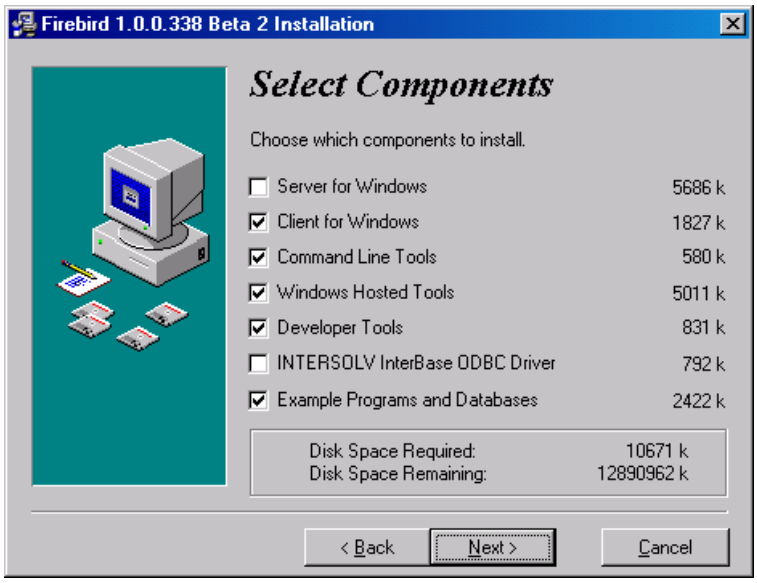


Рисунок 1.1. Выбор компонентов при установке Firebird

Установка InterBase на платформе Linux/Unix

Установка InterBase под Linix немного сложнее, чем на Windows, если вы не являетесь знатоком этой ОС. Для Linux InterBase (а также Firebird) существуют два варианта архитектуры сервера – SuperServer и Classic. Про их различия, а также про достоинства и недостатки поговорим позже, в главе "Classic и SuperServer", а пока будем считать, что речь идет о сервере с архитектурой SuperServer.

Инсталляционный пакет под Linux бывает двух видов – в формате rpm (для некоторых дистрибутивов Linux типа RedHat) и tar.gz (более универсальный пакет). Что это такое, можно узнать в любом приличном руководстве по Linux/Unix. Скачайте инсталляционный пакет того или иного вида (например, с сайта *www.firebirdsql.org*) и можно приступать к установке.

Для того чтобы произвести установку, необходимо выполнить следующие условия:

- Необходимо войти в систему как пользователь root.
- Для варианта Firebird SuperServer необходимо добавить в файл /etc/hosts.equiv строку вида localhost 127.0.0.1. Если такого файла нет, то его необходимо создать.
- Если необходимо получать доступ к установленному серверу с удаленных машин, то надо прописать имена и IP-адреса этих машин в файле hosts.equiv или позаботиться о разрешении имен с помощью DNS.

Для установки из rpm пакета необходимо выполнить следующую команду:

```
$rpm –Uvh InterBase.x.x.x.rpm
```

где InterBase.x.x.x.rpm является именем скачанного инсталляционного пакета.

Для установки с использованием пакета в формате tar.gz необходимо выполнить следующее:

```
$tar –xzf InterBase.x.x.x.tar.gz
$cd install
$./install.sh
```

Выполнение этих команд приводит к одному и тому же результату – на Linux-машине будет установлен InterBase (Firebird для нашего примера). Вариант с архитектурой SuperServer будет функционировать в виде демона, а Classic – в виде сервиса (см. главу "Classic и SuperServer" (ч. 4)).

Надо заметить, что приведенный порядок установки годится практически для любой версии InterBase 6.x/Firebird 1.0. Правда, для различных дистрибутивов Linux могут существовать свои особенности и хитрости при установке, так что лучше посмотреть рекомендации по установке для вашей версии ОС Linux/Unix.

Во время процесса установки останавливается любая предыдущая версия InterBase/Firebird, если она существует, конечно. Затем эта предыдущая версия архивируется (для целей резервного копирования, чтобы не потерять ценную информацию из-за ошибки или забывчивости).

Установка основного программного обеспечения производится в каталог /opt/InterBase, а библиотек и заголовков – в каталоги /usr/InterBase и /usr/include соответственно. В процессе установки изменяется пароль SYSDBA, причем каждый вид установочного пакета делает это по-разному: rpm создает случайный пароль и помещает его в файл /opt/InterBase/SYSDBA.password, а tar.gz запрашивает пароль в процессе установки.

Интересный вопрос – как запускается InterBase/Firebird под Linux? Вариант с архитектурой Classic запускается через inetd, как только поступает входящий запрос на соединение на порт, к которому привязан InterBase/Firebird (для InterBase 4.x, 5.x – только 3050, для Firebird – по умолчанию 3050, однако можно изменить это значение), inetd запускает новый экземпляр сервера и передает ему управление. Если нет соединений, то ничего и не запущено.

Вариант InterBase с архитектурой SuperServer прописывает стартовый скрипт в /etc/rc.d/init.d/Firebird и запускается в виде демона.

Для проверки инсталляции под Linux необходимо сначала проверить локальное соединение:

```
$cd /opt/InterBase/bin
$isql –user sysdba –password <password>
```

```
>connect  /opt/InterBase/examples/employee.gdb;
>select * from sales;
>quit;
```

Затем необходимо протестировать удаленное подключение:

```
$cd /opt/InterBase/bin
$isql –user sysdba –password <password>
>connect <hostname>:/opt/InterBase/examples/employee.gdb;
>select * from sales;
>quit;
```

Если выполняются запросы к Sales, то все в порядке – InterBase/Firebird работает.

Установка инструментария для администрирования InterBase

С InterBase всегда поставляются средства администрирования командной строки. Это очень мощные средства, которые мы будем постоянно применять для работы с примерами в этой книге. Однако пользователи привыкли использовать инструменты с графическим интерфейсом пользователя.

Вместе с InterBase поставляется один инструмент администрирования – IBConsole. К сожалению, этот инструмент недостаточно надежен и удобен, чтобы пользоваться им для администрирования InterBase и тем более для разработки баз данных.

К счастью, существует множество отличных инструментов от сторонних разработчиков (т. е. не работающих в компании Borland), которые удовлетворяют самым изысканным потребностям администратора и разработчика InterBase.

Среди самых известных и популярных можно перечислить IBExpert (www.ibexpert.com), IBManager (www.ibmanager.com), IBWorkbench и IBAdmin. Часть из этих продуктов являются свободно распространяемым (freeware), т. е. они полностью бесплатные, другая часть – условно бесплатные (shareware), т. е. имеется возможность использования ознакомительной версии. Со списком различных полезных инструментов и их краткими характеристиками можно ознакомиться в приложении "Инструменты администратора и разработчика InterBase".

Выберите инструмент по своему вкусу. Ряд этих продуктов разрабатывается российскими программистами, которые, понимая экономическую ситуацию в РФ и ближнем зарубежье, позволяют коллегам из бывшего СССР пользоваться своими инструментами бесплатно! Поэтому обязательно воспользуйтесь предоставленной возможностью и обзаведитесь удобным инструментом для работы с InterBase.

Все примеры в книге рассчитаны и на использование стандартного инструмента isql или других утилит командной строки, если это не оговорено особо. Но зачем делать что-то неудобным способом, когда есть прекрасная возможность избежать рутинной работы и потратить время на творчество?

Ключевые моменты установки

Итак, перечислим основыне ключевые моменты установки:

1) Остановить сервер InterBase/Firebird, если он запущен на компьютере.
2) Удалить все версии gds32.dll с данного компьютера.
3) Запустить инсталлятор и следовать его указаниям
4) Выбрать желаемую конфигурацию сервера.

Заключение

Теперь, когда на вашем рабочем компьютере установлен сервер и клиент какого-либо клона InterBase и мы обзавелись удобным инструментом для его администрирования, можно приступить к разработке базы данных на InterBase.

Создаем базу данных

Итак, нам необходимо создать новую базу данных. Это проще всего сделать при помощи какой-либо программы администрирования (например, IBExpert: http://www.blazetop.com).

Надо отметить, что абсолютно аналогичный процесс создания базы данных можно реализовать в любом другом менеджере InterBase.

Запустите BlazeTop, затем выберите в меню "Файл\Новый\Сервер". На экране появится диалог регистрации сервера InterBase. Он изображен на рис. 1.2.

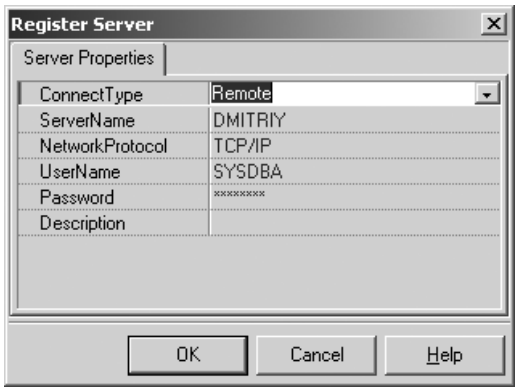


Рис. 1.2. Диалог регистрации сервера InterBase

Укажите тип подключения к серверу. Он может быть двух типов: локальным или удаленным. Имя сервера – это название компьютера в сети, на котором установлен InterBase. Укажите также сетевой протокол, который используется для подключения к серверу. По умолчанию это TCP/IP, а значит, в качестве названия сервера вы можете указывать прямой IP-адрес компьютера с установленным InterBase. Имя для подключения к серверу – это имя администратора базы данных. Для InterBase это всегда SYSDBA, это имя нельзя изменять. Однако настоятельно рекомендуется изменить пароль для SYSDBA. По умолчанию это ‘masterkey’. После регистрации сервера мы можем создавать базу данных. Выберите в меню "Файл\Новый\База данных". Появится диалог регистрации/создания базы данных (рис. 1.3).

Строка соединения

В свойстве ServerName укажите имя сервера из списка зарегистрированных (очевидно, что вы можете иметь несколько серверов на разных компьютерах). DatabaseName – это локальный путь к файлу базы данных на сервере. Если вы хотите зарегистрировать новую базу данных в BlazeTop, вы также можете присвоить базе какой-либо псевдоним при помощи свойства AliasName. Для собственно создания базы данных это не является необходимым. Укажите также дополнительные свойства: PageSize (размер страницы базы данных), CharSet (кодировка или кодовая страница по умолчанию для строковых полей), SQLDialect

(только для InterBase 6.x и Firebird). Вы можете также создать базу данных от имени любого пользователя, зарегистрированного на сервере.

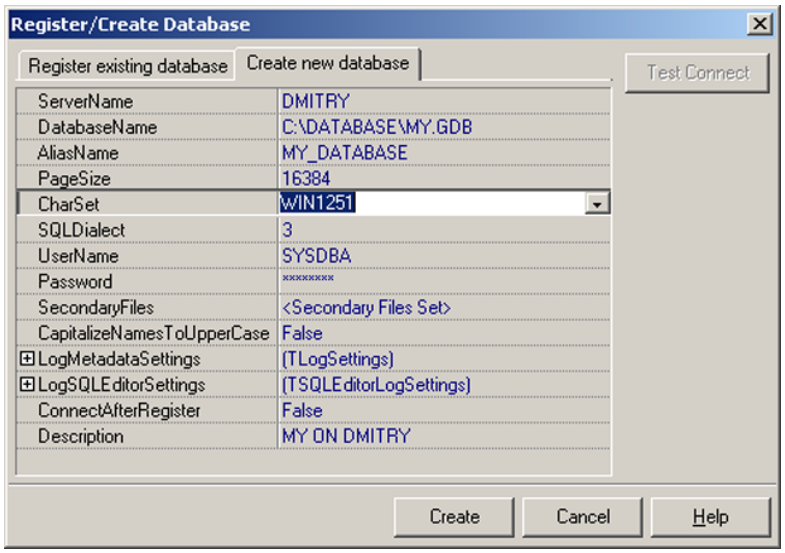


Рис. 1.3. Диалог создания базы данных

По умолчанию это, конечно, SYSDBA. Теперь BlazeTop сам создаст необходимый DDL-код для создания базы данных с теми свойствами, которые вы указали. Сначала он сформирует полный путь к базе данных, который называется **строкой соединения (connection string)**.

Вы уже знаете, что создать базу данных при помощи BlazeTop можно как на локальном компьютере, так и на удаленном. Если сервер InterBase находится над машиной под управлением какой-либо версии Windows (именно этот случай мы рассматривали в главе "Установка InterBase" в качестве типичного), а в качестве сетевого протокола установлен TCP/IP, то формат строки соединения, в котором указывается путь к вновь создаваемой базе и имя файла этой базы данных, будет следующий:

```
<имя_компьютера>:<Путь на компьютере\имя_базы_данных.gdb>
```

Несколько важных замечаний:

1. Формат строки соединения для других сетевых протоколов здесь не рассматривается – для получения информации о том, как сконфигурировать InterBase для работы по другим протоколам, обратитесь к разделу документации Operation Guide – Network configuration.

2. В данном примере рассматривается случай, когда InterBase-сервер (и база данных соответственно) находится на компьютере под управлением ОС Windows. В этом случае путь к базе данных начинается с буквы диска и каталоги разделяются обратной косой чертой. В случае, если InterBase-сервер и база данных расположены на машине под управлением *nix, то <путь на компьютере> будет начинаться с прямой косой черты и выглядеть примерно так: `/opt/database/firstbase.gdb`.

Например, если мы создаем базу данных с именем firstbase.gdb у себя на компьютере в каталоге C:\Temp, то строка соединения будет выглядеть следующим образом:

```
localhost:C:\temp\firstbase.gdb
```

Здесь localhost – имя компьютера, на котором создается база, C:\Temp – путь к вновь создаваемой базе данных, firstbase.gdb – имя базы данных. Localhost – это имя, зарезервированное для текущего (т.е. локального, Вашего, того, на котором запускаете программу) компьютера. Если понадобится создать базу данных на удаленном компьютере, например, server_nt, где-нибудь в каталоге C:\database, то путь будет, соответственно:

```
server_nt:C:\database\firstbase.gdb
```

Если на компьютере server_nt не будет каталога C:\Database, то вы получите ошибку. Также ошибка создания возникнет, если на server_nt не запущен (или вообще не установлен) InterBase – т. е. просто некому будет обработать запрос на создание базы данных. Момент создания и подключения к базе данных обычно вызывает массу проблем у новичков – они не сразу понимают, как правильно составить строку соединения. Запомните, что надо указывать путь к базе данных на том компьютере, где она находится (или будет находиться, если мы создаем базу).

> Помните, что имена netbios sharename, присваиваемые каталогам, отданным в совместное использование, никакого отношения к <пути_на_компьютере> не имеют. Также нет необходимости (и очень нежелательно с точки зрения безопасности) предоставлять разрешения на доступ к файлам баз данных пользователям.

Диалект базы данных

Вернемся к окну создания базы данных, изображенному на рисунке 1.3. Выбор диалекта базы данных очень важен. Свойство может принимать только два возможных значения: 1 или 3. Какое же выбрать?

Диалект 1 и Диалект 3 отличаются друг от друга следующими принципиальными вещами:

- Диалект 3 позволяет использовать расширенный набор типов данных, таких, как типы для работы с большими целыми числами, типы для работы с датой и временем – DATE и TIME.
- Диалект 3 различает регистр идентификаторов, **если** идентификатор заключен в двойные кавычки. Table1 и TABLE1 в обоих диалектах будут равнозначны, а вот “Table1” и “TABLE1” (TABLE1) – сервер будет интерпретировать как разные идентификаторы.
- Диалект 3 не поддерживает неявное приведение типов данных (как это было в Диалекте 1). Это означает, что в Диалекте 1 выражение ‘25’+5 будет корректным и в результате мы получим 30. В Диалекте 3 это выражение вызовет ошибку несоответствия типов.

Помимо этого, есть еще ряд отличий. Например, Borland Database Engine (BDE) вплоть до версии 5.2 плохо работает с 3-м диалектом – возникают проблемы с поддержкой новых типов данных.

- Выбор SQLDialect, в котором будет создаваться база данных, важен также по той причине, что переход между разными диалектами – занятие достаточно нетривиальное и трудоемкое, которого по возможности следует избегать. Иными словами, если есть возможность, то лучше сразу выбрать верный диалект.

В общем случае надо руководствоваться следующими правилами:

- выбираем Диалект 3, если мы будем проектировать базу данных для приложения, которое будет использовать только современные библиотеки прямого доступа к InterBase, которые полностью поддерживают Диалект 3;
- выбираем Диалект 1, если нам важна совместимость с более ранними библиотеками доступа к InterBase, такими, как BDE.

Может возникнуть вопрос, почему диалекты бывают 1 и 3, а где же 2? Диалект 2 действительно существует, но используется в качестве промежуточного этапа при миграции с Диалекта 1 на Диалект 3.

Размер страницы

Выбор размера страницы очень важен для обеспечения эффективной работы сервера InterBase с базой данных. Файл базы данных разбивается на страницы фиксированного размера, и все обращения к диску, которые выполняет InterBase, считывают и записывают информацию постранично. Выбирать следует размер страницы не менее 4096 байт, а лучше еще больше. Почему так? Дело в том, что если установить малый размер страницы, то записи большой длины (например, представьте себе 10 строковых полей в таблице, заполненных строками размером в 255 символов) будут занимать несколько страниц, и для чтения единственной записи InterBase будет вынужден осуществить несколько обращений к диску! Очевидно, что это не лучшим образом скажется на быстродействии.

Изменить размер страницы можно будет и позже, в процессе восстановления базы данных из резервной копии, но лучше все делать сразу и правильно, не так ли?

Рекомендации по выбору размера страницы следующие:

- Для дисковых накопителей с файловой системой NTFS выбираем размер страницы, равный 4096 байтам. Перед этим следует убедиться, что размер кластера у NTFS-диска установлен в 4096 байт (если не знаете, что такое кластер, спросите у вашего системного администратора).
- Для дисков с FAT32 (думаю, что FAT16 редко используется в наш век дешевых гигабайтов) устанавливаем размер страницы 8192 или 16384 байта (хотя стоит заметить, что размер страницы 16384 байта есть далеко не во всех клонах InterBase).

Кодировка (CharSet)

Кодировка также очень важна для базы данных. Кодировка определяет, символы какого национального алфавита будут использоваться в базе данных по умолчанию. Если вы предполагаете работать только с русским и английским языком, то лучше всего выставить кодировку WIN1251. Если же необходимо хранить информацию в базе на разных языках, то можно оставить значение ко-

дировки как NONE, а уже в настройках драйверов для библиотеки доступа определять необходимые опции для поддержки того или иного языка.

Как уже сказано, определение кодировки при создании базы данных задает набор используемых символов только **по умолчанию**. Во время создания таблиц, представлений и других объектов базы данных можно устанавливать другой набор символов, указав его явным образом.

Для большинства приложений баз данных вполне достаточно указать кодировку WIN1251 для всей базы данных по умолчанию.

Имя пользователя и пароль

Для создания базы необходимо указать имя пользователя и его пароль. Этот пользователь будет владельцем создаваемой базы данных (OWNER). Это дает ему право полностью управлять базой данных – создавать и удалять различные объекты базы данных, выполнять запросы на выборку и изменение данных – в общем, быть ее полным хозяином.

Обычно для создания базы применяют права встроенного пользователя InterBase – SYSDBA. Этот пользователь реализует функции системного администратора базы данных. По умолчанию SYSDBA может изменять все объекты базы данных, вне зависимости от того, кем они созданы – самим SYSDBA или другим пользователем.

Конечно, очень удобно иметь полный набор прав "в одном флаконе", когда разрабатываешь базу данных, но во время промышленной эксплуатации часто требуется разделить права между различными группами пользователей. Вот тогда применяются пользователи (USERS) и роли (ROLES) для того, чтобы на уровне базы данных разграничить доступ к информации, хранящейся в базе данных. Подробнее о безопасности в InterBase см. главу "Безопасность в IInterBase: Пользователи, роли и права" (ч. 4).

Рекомендованный пользователь для создания базы данных – SYSDBA. Это позволяет на этапе начального проектирования базы не беспокоиться о распределении прав – этим можно будет заняться несколько позже. Пароль для SYSDBA по умолчанию – ‘masterkey’. Конечно, этот пароль лучше сменить, особенно на серверах, находящихся в промышленной эксплуатации.

Что получилось

В данном примере мы создаем базу данных InterBase с именем 'my.gdb' в каталоге 'C:\Database' на локальном компьютере. Размер страницы создаваемой базы данных равен 16384 байтам, в качестве кодировки по умолчанию выбрана WIN1251 – набор символов, включающих кириллицу. Выберем для базы SQLDialect 3.

Менеджеры InterBase/Firebird предоставляют очень удобный интерфейс для создания базы данных, однако надо сказать, что создание базы данных "вручную" не намного сложнее. Для этого достаточно создать два файла. Первый из них – это файл с командами SQL (его называют файлом *скрипта*), который создаст базу данных, второй – командный файл WINDOWS, который передаст этот SQL-файл на выполнение утилите isql.exe. Эта утилита выполнит выполнит

скрипт создания базы данных. Вот содержимое этих двух файлов для нашего случая:
файл скрипта crebas.sql:

```
SET SQL DIALECT 3;
SET NAMES WIN1251;
CREATE DATABASE 'Localhost:C:\Database\my.gdb'
USER 'SYSDBA' PASSWORD 'masterkey'
PAGE_SIZE 16384
DEFAULT CHARACTER SET WIN1251;
```

командный файл runscr.bat:

```
"C:\IBServer\Bin\isql.exe" -i "C:\temp\crebas.sql"
```

Естественно, в командном файле должны быть прописаны реальные пути к isql.exe (это интерпретатор команд SQL, входящий в комплект InterBase) и к нашему файлу скрипта crebas.sql.

Типы данных

Несмотря на то, что типы данных подробно описаны в документации (см. [1, гл. 4]), необходимо рассмотреть ряд понятий, которые будут часто использоваться в последующих главах книги. Помимо изложения сведений общего характера будут рассмотрены также примеры использования типов данных в базах данных InterBase и изложены рекомендации по их использованию и преобразованию. Также подробно рассмотрим отличия в типах данных, существующие 1-м и 3-м диалектах базы данных InterBase.

О типах данных

Типы данных – это базовые элементы любого языка программирования или любого сервера СУБД, и InterBase не исключение. Когда мы говорим, что в базе данных хранится какая-то информация, то должны всегда четко осознавать, что эта информация не может быть свалена в одну большую кучу; наоборот, данные должны быть рассортированы и разложены по "полочкам". Типы данных определяют, что можно положить на соответствующую "полочку", а что нельзя. Под "полочками" понимаются прежде всего поля таблиц в базе данных (см. главу "Таблицы. Первичные ключи и генераторы" (ч. 1)), а также переменные внутри триггеров, хранимых процедур и т. д.

Каждый тип данных имеет набор операций, которые можно выполнять над значениями этого типа, поэтому необходимо правильно выбрать тип данных при проектировании базы данных, что поможет избежать многих проблем при разработке клиентских программ.

В InterBase существует 12 типов данных, которые способны удовлетворить практически любые потребности разработчика в хранении данных. Эти типы условно подразделяются на 6 следующих групп:

- для хранения целых чисел – INTEGER и SMALLINT;
- для хранения вещественных чисел – FLOAT и DOUBLE PRECISION;
- для чисел с фиксированной точностью – NUMERIC и DECIMAL;
- для хранения даты, времени и даты/времени – DATE, TIME и TIMESTAMP;
- для хранения символов – CHARACTER (сокращенно CHAR) и VARYING CHARACTER (VARCHAR);
- Для хранения динамически расширяемых данных – BLOB.

Также возможно определять массивы значений элементарных типов, т.е. всех перечисленных типов, кроме BLOB.

Большинство типов данных InterBase соответствуют типам, определенным в стандарте SQL92, однако, помимо этого, есть и собственные "изюминки" – массивы элементарных типов данных и BLOB.

Массивы в InterBase могут содержать множество данных одного типа в одном поле, например можно определить массив значений типа INTEGER. Причем массивы могут иметь несколько размерностей!

Тип данных BLOB – это динамически расширяемый тип данных, название которого часто расшифровывается как Binary Large OBject – "большие двоичные объекты". Надо сказать, что BLOB – это изобретение разработчиков InterBase, которое позже распространилось и прижилось во всех современных SQL-серверах.

Синтаксис определения типов данных

Типы данных используются для описания полей в таблицах, переменных в триггерах и хранимых процедурах. Ниже представлен общий синтаксис определения всех возможных в InterBase типов данных.

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

Подробно свойства типов данных, такие, как размер, точность и диапазон возможных значений, описаны в табл. 4.1 в [1], поэтому повторяться здесь не будем. Далее кратко рассмотрим основные особенности типов данных и сосредоточимся на их возможном применении.

Целочисленные типы

К целочисленным типам относятся SMALLINT и INTEGER. Надо сказать, что SMALLINT представляет собой урезанную версию INTEGER и имеет длину 2 байта, в отличие от 4 байт, выделяемых для хранения INTEGER. Обычно экономить на дисковом пространстве не следует, и поэтому общей рекомендацией будет использовать для хранения целых значений тип INTEGER.

Область применения целочисленных типов очевидна: они нужны для полей, содержащих только целые числа – для хранения счетчиков, количества и т.д. Обычно тип INTEGER имеют также поля, содержащие первичные ключи.

Вещественные типы данных

К вещественным типам (их еще называют типами чисел с плавающей точкой) относятся FLOAT и DOUBLE PRECISION. Сразу следует предостеречь читателя от использования типа FLOAT – его точность недостаточна для хранения большинства дробных значений. Особенно не рекомендуется хранить в нем денежные величины – в переменных типа FLOAT очень быстро нарастают ошибки округления, что может сильно удивить бухгалтера при подведении итогов.

Если в базе данных предполагается хранить числа с плавающей точкой (например, в бухгалтерских системах или в системах для научных расчетов), то лучшим выбором будет тип DOUBLE PRECISION.

> Надо отметить, что в 3-м диалекте InterBase для хранения денежных величин существует механизм хранения типов с фиксированной точкой длиной 64 бита. Использование этих типов обеспечивает наилучшую точность.

Типы данных с фиксированной точкой

К этим типам данных относятся NUMERIC и DECIMAL. Часто звучит вопрос, чем NUMERIC отличается от DECIMAL. Оба этих типа имеют одинаковую разрядность – от 1 до 18 знаков, одинаковую точность – от нуля до разрядности.

> Напомним, что: *разрядность* – это общее число цифр в числе, а *точность* – число знаков после запятой.

Самое забавное, что, несмотря на то что в документации написано, что эти типы отличаются максимальной разрядностью, на самом деле реализованы они практически одинаково и разницы между ними никакой нет! Вы легко можете это проверить, запустив утилиту isql и произведя нижеследующую очередность действий.

Создаем таблицу следующего вида:

```
SQL> CREATE TABLE test (
CON> Num_field NUMERIC(15,2),
CON> Dec_field DECIMAL(15,2));
```

Затем даем команду показать структуру таблицы:

```
SQL> show tables test;
```

И наблюдаем такую картину:

```
NUM_FIELD                       NUMERIC(15, 2) Nullable
DEC_FIELD                       NUMERIC(15, 2) Nullable
```

Как видите, InterBase сообщает о том, что оба данных столбцы имеют тип NUMERIC!

Причины такого поведения лежат в реализации типов данных с фиксированной точкой. Дело в том, что InterBase имеет всего 3 механизма хранения любого целочисленного выражения, и все типы, как бы они ни назывались, приводятся к этим вариантам реализации.

Вот таблица из [1], которая иллюстрирует, как хранятся различные целочисленные типы (табл. 1.1). Как видите, хранение данных в 3-м диалекте отличается для чисел с большой разрядностью:

Таблица 1.1. Хранение чисел с фиксированной точкой

Разрядность	Диалект 1	Диалект 3
От 1 до 4	SMALLINT для NUMERIC INTEGER для DECIMAL	SMALLINT
От 5 до 9	INTEGER	INTEGER
От 10 до 18	DOUBLE PRECISION	INT64

Итак, теперь мы точно можем сказать, чем отличаются типы NUMERIC и DECIMAL: в случае определения поля (переменной) с малой разрядностью

(до четырех) первый хранится в виде 2 байтового целого числа SMALLINT, а второй – в виде 4 байтового INTEGER.

Таким образом, в случае разрядности, большей четырех, типы DECIMAL и NUMERIC окажутся абсолютно эквивалентными!

Обратите внимание на отличие реализации типов с большой разрядностью в 1-м и 3-м диалектах. В 1-м диалекте число с фиксированной точкой превращалось из целого в вещественное, к которому применялись механизмы округления! В 3-м диалекте эта странность была ликвидирована – большие целые числа хранятся действительно как целые – с использованием механизма INT64, который может хранить 64-битовые числа в диапазоне +/– 2^32. Поэтому рекомендуется хранить данные о денежных средствах в базах данных, созданных с использованием 3-го диалекта, – только при использовании механизма INT64 можно гарантировать сохранность малых денежных остатков.

Типы для хранения даты и времени

Типы для хранения даты и времени изменились в версии InterBase 6.x и его клонах по сравнению с 4.x и 5.x. Чтобы не путаться в исторических хитросплетениях с этими типами, рассмотрим ситуацию именно в 6-й версии InterBase, а затем на основе этого кратко упомянем о том, что было раньше, – это делается для тех пользователей, кто все еще работает на ранних версиях InterBase

Итак, в InterBase 6.x существует 3 типа для хранения даты и времени – это DATE, TIME и TIMESTAMP.

- Тип DATE хранит даты с точностью до дня. Диапазон возможных значений – от 1 января 100 года н. э. до 29 февраля 32768 года.
- Тип TIME хранит данные о времени с точностью до десятитысячной доли секунды. Диапазон возможных значений – от 00:00 AM до 23:59.9999 PM.
- Тип TIMESTAMP представляет собой комбинацию типов DATE и TIME.

Как работать с датами? Если речь идет о работе на уровне сервера в хранимых процедурах или триггерах, то все достаточно просто – мы всегда можем объявить переменную нужного нам типа и присваивать ей значения из таблиц и наоборот. Однако необходимо передавать данные из базы данных в приложение и обратно. В этом случае есть два подхода – либо использовать библиотеки, которые применяют оригинальный формат дат InterBase для доступа к объектам этих типов и преобразуют этот формат в привычные внутриязыковые типы даты/времени (примером такой библиотеки является FIBPlus), либо использовать механизм преобразования дат в строки, встроенный в InterBase.

Что делать, если нужно вырезать из полной даты только год или месяц? Для этого используется группа функций EXTRACT (доступная во всех клонах InterBase 6.x), которая позволяет выделить из даты только нужную часть. Используются эти функции следующим образом:

```
EXTRACT (MONTH FROM DATE_FIELD)
EXTRACT (YEAR FROM DATE_FIELD)
```

Полный список параметров в функции EXTRACT таков: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, WEEKDAY, YEARDAY. Их назначение очевидно следует из их названия, поэтому не будем приводить здесь расшифровки.

Типы данных для хранения текста

В InterBase существует два типа, предназначенных для хранения текстовой информации – CHAR и VARCHAR. Полные их названия, – CHARACTER и CHARACTER VARYING, однако нет никакой причины пользоваться длинными именами – даже команда Show tables в утилите isql выдает краткие наименования типов.

Чтобы определить поле или переменную символьного типа, необходимо в скобках после имени типа либо указать число символов, которое будет использоваться в определяемом объекте, либо опустить число символов – при этом будет создано поле с длиной 1 символ.

```
CREATE TABLE testCHARLen(
Field1 CHAR(255),
Field2 CHAR);
```

В результате создания этой таблицы поле Field1 будет иметь длину 255 символов, а Field2 – 1 символ.

Типы CHAR и VARCHAR во многом схожи – оба могут содержать до 32768 символов, однако есть и отличия. Хотя хранятся эти два типа в базе данных одинаково, но работает с ними InterBase по-разному. Это можно продемонстрировать следующим примером:

```
SQL> create table testCHAR ( c1 char(10), c2 varchar(10));
SQL> insert into testCHAR(c1,c2) values('Test','Test');
SQL> SELECT '('||c1||')', '('||c2||')' from testCHAR;
```

В результате получим следующий результат:

```
(Test      ) (Test)
```

Как видите, после значения 'Test', выбранного из поля c1, оказались пробелы. Это означает, что при выборке данных из поля типа CHAR возвращаемое значение дополняется пробелами до полной длины поля. Сложно предположить, для чего необходимо подобное поведение, которое приводит к значительному увеличению сетевого трафика (загрузки сети).

В любом случае рекомендованным к использованию символьным типом является VARCHAR.

Одной из важнейших характеристик символьного типа является его *набор символов* – CHARACTER SET. Набор символов определяется для всей базы данных и используется по умолчанию для всех символьных полей, если не переопределяется явно при создании поля.

Чтобы создать символьное поле с явным указанием набора символов, необходимо в описании столбца (в предложениях CREATE TABLE или ALTER TABLE) добавить описание набора символов. Для поддержки русского языка обычно используется набор символов WIN1251 (подробнее об использовании русского языка в InterBase см. главу "Русификация InterBase" (ч. 1)). Вот пример таблицы, содержащей символьное поле с явно описанным набором символов WIN1251:

```
CREATE TABLE TestCHARSET(
Field1 VARCHAR(255),
Field2 VARCHAR(255) CHARACTER SET win1251);
```

Здесь Field1 – поле без явного указания набора символов, поэтому для него будет использоваться тот набор символов, который был указан при создании базы данных. Для поля Field2 явно определено, что в нем будут храниться символы в кодировке WIN1251.

Помимо указания набора символов, для символьных полей возможно также указывать порядок сортировки (COLLATION ORDER), который определяет, как будут сортироваться символы этого набора данных. Для русского языка существуют два варианта сортировки – WIN1251 и PXW_CYRL. Подробнее об использовании COLLATION ORDER рассказано в главе "Русификация InterBase".

Полный список наборов символов и применяемых для них COLLATION ORDER можно найти в документации [1, гл. 13].

Внимание! В документации на InterBase 6 сказано, что символьных типов 4: помимо указанных выше типов данных существуют еще NCHAR и NCHAR VARYING, однако ниже в той же документации объясняется, что последние два типа являются теми же типами CHAR и VARCHAR, только используют по умолчанию набор символов ISO8859_1. То есть фактически использование псевдотипа NCHAR равносильно применению CHAR DEFAULT CHARACTER SET ISO8859_1. Аналогично и для NCHAR VARYING, только там вместо CHAR используется VARCHAR. Очевидно, что применение этих псевдотипов ориентировано на пользователей в Западной Европе и США, для поддержки языков в которых и создан набор символов ISO8859_1.

Тип данных BLOB

Тип данных BLOB предназначен для хранения большого количества данных переменного размера. Тип BLOB позволяет хранить данные, которые не могут быть помещены в поля других типов, – например, картинки, музыкальные файлы, видеофрагменты и т. д.

Чтобы определить самое простое поле типа BLOB в таблице, не нужно ничего сверх того, что обычно требуется для определения поля любого элементарного типа:

```
CREATE TABLE testBLOB(
myBlobField BLOB);
```

В результате будет создано поле myBlobField, в котором можно хранить данные большого размера. Но несмотря на то что поля BLOB по способу определения никак не отличаются от других, реализация их внутри базы данных значительно отличается. Не-BLOB-поля расположены на странице данных (см. главу "Структура базы данных InterBase" (ч. 4)) рядом друг с другом, а в случае BLOB на странице данных хранится только идентификатор BLOB, а сам BLOB располагается на специальной странице. Именно такая организация данных позволяет хранить данные нефиксированного размера.

У типа BLOB имеется возможность определять набор нескольких подтипов и специальных процедур, называемых *фильтрами* (BLOB filters), для работы

с этими подтипами. Существует несколько предопределенных подтипов BLOB, которые встроены в InterBase. Все эти подтипы имеют неотрицательные номера, например subtype 0 – это данные неопределенного типа, subtype 1 – текст, subtype 2 – BLR (Binary Language Representation, см. глоссарий и главу "Структура базы данных InterBase") и т. д. Пользователь также может определять свои подтипы BLOB, которые могут иметь отрицательные значения. Каждому типу может быть поставлен в соответствие фильтр, который преобразует поле этого подтипа в другой подтип.

Надо отметить, что использование BLOB-полей обычно служит альтернативой хранению внешних относительно базы данных файлов. Что касается фильтров BLOB, то они используются достаточно редко по причине своей ориентации на узкий класс задач.

Массивы

СУБД InterBase была одной из первых, в которой появились массивы. Поддержка массивов в базе данных является расширением традиционной реляционной модели. Наличие массивов позволяет упростить работу со множествами данных одного типа.

Массив – это совокупность значений одного типа, имеющая общее имя и позволяющая обратиться к любому элементу массива по его номеру. Массивы в InterBase могут быть одномерными и многомерными.

Для того чтобы создать в таблице поле типа массив чисел INTEGER, необходимо написать что-то вроде следующего:

```
CREATE TABLE test(
myOneDimArray INTEGER[12],
myTwoDimArray INTEGER[5,4],
myThreeDimArray INTEGER[2,10,8]);
```

При этом создадутся 3 поля типа массив: поле myOneDimArray, содержащее одномерный массив длиной 12 чисел, myTwoDimArray, содержащее двумерный массив (матрицу) 5x4 чисел Integer, и поле myThreeDimArray – трехмерный массив 2x10x8. Надо отметить, что при таком определении элементы массива нумеруются начиная с единицы, т. е. первый элемент имеет номер 1, второй – номер 2 и т. д. Если кто-то хочет указать границы массива самостоятельно, например с 0 до 5, то он должен задать определение поля так:

```
myArray INTEGER[0:5]
```

Массивы реализованы на базе полей типа BLOB, поэтому не следует опасаться, что многомерный массив "загрязнит" вашу таблицу невероятным количеством данных: InterBase аккуратно разместит данные массива на отдельных страницах, чтобы оптимизировать операции ввода-вывода в этих полях.

Как использовать массивы? Они предоставляют удобный механизм для хранения однотипных объектов. Однако в 80 % случаев вместо массивов разработчики предпочитают держать множественные данные в подчиненных (detail) таблицах, поэтому массивы не так часто используются в клиентских приложениях СУБД InterBase. Этому немало способствует то, что поставляемые в комплекте с Delphi и C++Builder библиотеки доступа, такие, как BDE и IBX, не имеют воз-

можности работать с массивами. В документации по InterBase упоминается о возможности работать с массивами с помощью препроцессора gpre, однако это не самый удобный способ для разработчика Delphi/C++Builder. К счастью, в библиотеке FIBPlus имеется поддержка полей-массивов в InterBase, о чем подробно рассказано в главе "Специальные возможности FIBPlus". Клиентская библиотека IBProvider, позволяющая создавать клиентские приложения для InterBase с помощью средств разработки компании Microsoft, также поддерживает работу с массивами (см. главу "Разработка клиентских приложений СУБД InterBase с использованием технологии Microsoft OLE DB" (ч. 3)).

Заключение

Надо отметить, что невозможно рассказать о типах данных, не забегая вперед, – настолько они проникают во все ключевые области, связанные с разработкой приложений баз данных. Поэтому в процессе чтения этой книги стоит использовать данную главу как справочник, к которому можно обращаться всякий раз, когда надо освежить в памяти основы InterBase.

Таблицы. Первичные ключи и генераторы

InterBase – это реляционная СУБД. Помимо всего прочего это означает, что все данные в InterBase хранятся в виде таблиц. Таблица, как ее понимают с точки зрения SQL, очень похожа на обычную таблицу, которую можно нарисовать от руки на листе бумаги или создать в программе вроде Microsoft Excel. У таблиц в InterBase имеются столбцы и строки, в которых размещаются данные. Таблица обязательно имеет имя, уникальное в пределах одной базы данных. Таблицы являются основным хранилищем информации в базе данных, и поэтому необходимо ответственно относиться к созданию таблиц.

Существуют правила, описывающие, как создавать таблицы в реляционной базе данных, отражающие данные реального мира и в то же время позволяющие организовать эффективное хранение информации в базе данных. Процесс применения этих правил для проектирования "правильной" базы данных называется *нормализацией*. Мы не зря взяли слово "правильной" в кавычки, потому что "нормализованная база данных" и "оптимизированная база данных" не являются синонимами. Необходимо не просто слепо следовать правилам нормализации, но и всегда делать поправку на условия конкретной задачи.

Нормализация таблиц в базе данных хорошо и подробно рассмотрена в книге [14], и потому мы не будем пытаться объять необъятное и вернемся к нашей конкретной области – к таблицам InterBase. Рассмотрим синтаксис предложения DDL (DDL – это Data Definition Language, подробнее см. в глоссарии), которое позволяет создавать таблицы:

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

Здесь table – имя создаваемой таблицы, <col_def> – описание столбцов (иногда мы будем говорить - полей) создаваемой таблицы. Опция table [EXTERNAL [FILE] “<filespec>”] означает, что будет создана так называемая внешняя таблица, которая хранится не в общем файле базы данных, а в отдельном файле с именем <filespec>.

Как видите, все просто – определяем имя таблицы и столбцы, которые в нее входят. Теперь надо подробнее рассмотреть, как определить столбцы. Синтаксис создания столбца описывается следующим предложением DDL:

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

Довольно большое определение, однако лишь небольшая часть приведенных в определении столбца предложений является обязательной. Каждый столбец в таблице должен иметь имя, уникальное в пределах таблицы, а также либо тип данных, определяемый предложением datatype, либо выражение <expr> для вычисления значения столбца (для вычисляемых столбцов), либо домен (см. ниже), определяемый domain. Типы данных были рассмотрены в главе "Типы данных", поэтому вы легко можете понять, как формируется SQL-выражение для создания таблицы.

Давайте подключимся к нашей базе данных FIRSTBASE.gdb, созданной ранее в главе "Создаем базу данных", и попробуем поработать с таблицами на практике. Для создания, удаления и модифицирования таблиц подойдет как любой из административных инструментов InterBase – из тех, что перечислены в приложении "Инструменты администратора и разработчика InterBase", так и стандартная утилита isql.exe из комплекта поставки любого клона InterBase.

Вот пример простой таблицы, названной TABLE_EXAMPLE и содержащей 3 поля различных типов:

```
CREATE TABLE Table_example (
ID INTEGER,
NAME VARCHAR(80),
PRICE_1 DOUBLE PRECISION);
```

Эта таблица иллюстрирует наиболее часто встречающийся в процессе разработки базы данных случай.

Однако возможны и другие способы определения полей. Например, мы можем задать тип поля, используя домены. *Домен* – это тип, определяемый пользователем для удобства применения определенных сочетаний параметров типов. Например, можно определить домен D_ID для задания полей идентификаторов. Определив домен, можно воспользоваться им для задания типа поля:

```
CREATE DOMAIN D_ID AS INTEGER;
CREATE TABLE Table_example (
ID D_ID,
NAME VARCHAR(80),
PRICE_1 DOUBLE PRECISION);
```

При этом поле ID будет иметь тип, определяемый доменом D_ID. Таким образом, определив в домене тип поля, необходимые проверки и ограничения, мы можем многократно применять этот домен для создания полей одинакового назначения, например денежных, без утомительного и таящего в себе опасность ошибиться переписывания определений типов переменных.

Третий способ задать столбец в таблице – это определить его как вычисляемый (COMPUTED BY) и задать условие, согласно которому будет вычисляться его значение. Например, мы можем пожелать иметь в нашей таблице столбец, вычисляющий 10% от значения поля PRICE_1. Для этого мы можем ввести следующую команду:

```
CREATE TABLE Table_example (
ID INTEGER,
NAME VARCHAR(80),
PRICE_1 DOUBLE PRECISION,
PRICE_10 COMPUTED BY (PRICE_1*0.1));
```

Но не думайте, что теперь, как только мы вставим данные в поле PRICE_1, в поле PRICE_10 окажется десятая часть от значения этого поля. Нет, процесс тут более сложен. На самом деле мы получим искомую десятую часть только при обращении к полю PRICE_10, например при выполнении запроса SELECT к этой таблице. То есть в вычисляемом поле не хранятся какие-либо данные,

а при каждой выборке производится вычисление выражения, связанного с полем, и результат выдается в ответ на запрос.

Итак, мы рассмотрели 3 основных способа задания полей в таблице. Теперь давайте подробнее рассмотрим опции, которые можно задать при создании столбца.

Опция [DEFAULT {literal | NULL | USER}] – позволяет задать значение столбца по умолчанию. Это очень удобная возможность для автоматического заполнения данных. Существует 3 возможности задавать значения по умолчанию. Первая обозначена как literal и позволяет задавать значения по умолчанию в виде текстовых констант, чисел или даты. Например, мы можем сформировать следующие выражения для создания столбца с текстовыми значениями по умолчанию:

```
NAME VARCHAR(80) DEFAULT 'Василий Станиславович'
```

При этом все заносимые в таблицу поля будут принимать значения по умолчанию, т. е. если не было определено другого значения для поля NAME, то там появится строка 'Василий Станиславович'.

Вторая возможность задавать значения по умолчанию – это указать DEFAULT NULL в определении столбца. При этом во вновь создаваемых записях значение этого столбца будет NULL, если, конечно, не было явно задано иное значение. Пример:

```
PRICE_1 DOUBLE PRECISION DEFAULT NULL
```

И третий способ задать значение по умолчанию – это указать DEFAULT USER в определении столбца. При этом во вновь создаваемых записях в это поле будет заноситься имя текущего пользователя, т. е. пользователя, который установил соединение с InterBase и выполнил эту вставку (подробнее о пользователях см. главу "Безопасность в InterBase: пользователи, роли и права" (ч. 4)).

Для некоторых полей совершенно необходимо, чтобы поле имело какое-то непустое значение. Например, поле, которое по условиям задачи не может быть пустым. Чтобы на уровне базы данных задать ограничение о том, что поле должно иметь какое-то определенное значение, нужно внести следующее добавление в описание столбца:

```
NAME VARCHAR(80) NOT NULL
```

При этом создастся поле, в котором нельзя хранить неопределенные (NULL) значения. Обычно ограничение NOT NULL сочетается с опцией DEFAULT, которая гарантированно присваивает какое-либо корректное значение этому полю.

Но часто ограничения NOT NULL бывает недостаточно. Например, в случае хранения в базе данных каких-либо цен совершенно очевидно, что они не могут принимать отрицательные значения (хотя было бы здорово, чтобы нам приплачивали при покупке какого-нибудь товара). Чтобы заставить сервер проверять заносимые в базу данных значения цен на условие положительности, следует так определить столбец:

```
PRICE_1 DOUBLE PRECISION CHECK (PRICE_1>0)
```

При этом заносимые в столбец PRICE_1 значения будут проверяться на условие положительности. Надо отметить, что различные непротиворечивые опции

могут сочетаться, например можно задать непустое значение и проверку на положительность:

```
PRICE_1 DOUBLE PRECISION NOT NULL CHECK (PRICE_1>0)
```

Некоторые опции при создании столбцов сочетаться не могут, например нельзя задать значение по умолчанию NULL и одновременно ограничение на непустое значение.

Надо отметить, что проверки могут выполнять множество полезных функций по управлению данными в базе данных. Подробнее их использование мы рассмотрим в главе "Ограничения базы данных".

Итак, мы рассмотрели способы создания таблиц и полей с различными опциями.

Однако бывают случаи, когда необходимо изменить уже существующую таблицу. Конечно, мы можем пересоздать таблицу целиком. Для этого следует выполнить команду удаления таблицы и вновь создать ее, например так:

```
DROP TABLE Table_example;
CREATE TABLE Table_example(ID NUMERIC(15,2);
```

Но такой способ изменения таблиц имеет значительные недостатки. При удалении таблицы с помощью команды DROP все данные, которые существовали в таблице, уничтожаются и, чтобы их не потерять, приходится копировать их во временные таблицы. Это весьма хлопотно. Поэтому для легкого изменения структуры таблиц существует команда ALTER TABLE, которая позволяет добавлять новые поля, удалять существующие, а также добавлять/удалять ограничения ссылочной целостности.

Например, мы желаем добавить в таблицу еще один столбец, предназначенный для хранения данных об отчестве человека:

```
ALTER TABLE Table_example ADD Patronimic VARCHAR(80);
```

После выполнения этой команды в нашей таблице Table_example появится новый столбец с именем Patronimic и типом VARCHAR(80). А если мы пожелаем удалить из таблицы столбец с именем NAME, то следует выполнить

```
ALTER TABLE Table_example DROP Name;
```

Полный синтаксис предложения ALTER TABLE можно узнать в [1]. Это очень полезная команда, и мы будем часто ею пользоваться.

А что делать, спросите вы, если надо изменить сам столбец? Например, мы решили, что для хранения имен лучше использовать поле HUMAN_NAME, а не просто NAME. В этом случае мы можем применить команду ALTER TABLE <table> ALTER COLUMN, которая позволяет изменять наименование столбца.

Это новая команда, поддерживаемая только в InterBase 6.x и его клонах. В InterBase более ранних версий для изменения имени и любых параметров столбца пришлось бы создавать временное поле, удалять существующий столбец, создавать новый – с нужным именем и параметрами – и затем копировать данные из временного поля в новый столбец. А в 6-м InterBase это можно сделать одной командой, например такой:

```
ALTER TABLE Table_example
ALTER COLUMN NAME TO HUMAN_NAME;
```

Если же мы решили изменить тип поля, например увеличить число символов, хранимых в поле, то придется изменять домен этого поля, используя предложение ALTER DOMAIN (см. выше главу "Типы данных").

Итак, мы рассмотрели создание и модификацию таблиц в InterBase. Теперь придется немного углубиться в теорию баз данных. InterBase, как уже было сказано, является реляционной базой данных. Помимо всего прочего это означает, что каждая запись в таблице должна иметь некоторый признак, по которому одну запись можно отличить от другой. Для этой цели служит специальный механизм уникальных ключей.

Первичные ключи в таблицах

Конечно, мы можем создать таблицу, не содержащую никаких ключей. Это никто нам не запрещает. Но, как уже говорилось, создание работоспособной базы данных невозможно без следования правилам нормализации. Наличие ключей – важнейший элемент нормализации. Поэтому, хоть мы и не ставим целью на рассмотрение теории и нормализации баз данных, нам придется ввести определение ключей и рассмотреть их роль в InterBase.

Будем двигаться постепенно и начнем с самого распространенного типа ключей – с первичного ключа.

Итак, что же такое *первичный ключ*? Это одно или более полей в таблице, однозначно идентифицирующих записи в пределах этой таблицы. Звучит сложно, однако на самом деле все очень просто. Представьте себе обыкновенную таблицу, например бухгалтерскую ведомость. Что является самым первым столбцом? Правильно, порядковый номер – 1, 2, 3… Этот номер указывает на уникальную строку в пределах таблицы, и достаточно знать этот номер, чтобы найти в этой таблице строку. В данном примере это и будет первичный ключ.

Абсолютное большинство таблиц в реляционной базе данных обязательно имеют первичный ключ (часто пишут PK – сокращение от Primary key). Общей рекомендацией при создании таблиц является создавать первичный ключ. Создать первичный ключ можно как при создании таблицы, так и позже. Допустим, мы уже к моменту создания таблицы решили, что первичным ключом у нас будет поле ID. Тогда добавить первичный ключ можно следующим образом:

```
CREATE TABLE Table_example (
    ID INTEGER NOT NULL,
    NAME VARCHAR(80),
    PRICE_1 DOUBLE PRECISION,
    CONSTRAINT pkTable PRIMARY KEY (ID));
```

Итак, что необходимо сделать, чтобы создать первичный ключ на таблицу table_example. Внимательно рассмотрим, что изменилось в определении таблицы? Во-первых, колонка ID получила дополнительное определение NOT NULL. Это важно – первичный ключ должен быть уникальным и не допускать неопределенных значений. А NULL, как вы знаете, это неопределенное значение. Та-

ким образом, все поля, входящие в первичный ключ, должны иметь ограничение NOT NULL.

Чтобы завершить создание первичного ключа, в конце таблицы дописывается предложение вида CONSTRAINT <имя_ключа> <тип_ключа> (<поля_входящие_ в_ключ>). Полный синтаксис ограничений рассмотрен в главе "Ограничения базы данных" ч. 1, и для нашего примера первичного ключа будет иметь вид:

```
CONSTRAINT pkTable PRIMARY KEY (ID)
```

Здесь – pkTable – имя первичного ключа, а ID – столбцы, входящие в него.

Такой способ определять первичные ключи для таблиц удобен при массовом создании таблиц (например, при построении прототипа базы данных на основе скриптов, получаемых с помощью различных CASE-средств). Но что делать, если нам нужно добавить/удалить первичный ключ в таблицу, когда она уже существует и наполнена данными? Для этого следует воспользоваться еще одним расширением команды – ALTER TABLE. Пример добавления первичного ключа в нашу таблицу:

```
ALTER TABLE TABLE_EXAMPLE ADD CONSTRAINT FF PRIMARY KEY (ID);
```

При этом в таблицу Table_example добавится точно такой же первичный ключ, как и в предыдущем примере, когда он создавался вместе с таблицей. Чтобы удалить первичный ключ, необходимо ввести следующую команду:

```
ALTER TABLE Table_example DROP CONSTRAINT pkTable;
```

При этом ключ с именем pkTable будет удален из базы данных.

Генераторы – удобная основа для первичных ключей

Надо сказать несколько слов о реализации первичного ключа. Так как он предназначен для обеспечения уникальности, то никакие две записи в одной таблице не могут иметь одинаковых значений этого ключа. То есть, чтобы удовлетворить этому условию, при занесении новой записи в таблицу InterBase должен просмотреть все записи в таблице и выяснить, нет ли уже таких значений в таблице. Для быстрого поиска в InterBase существует механизм *индексов* – специальных объектов InterBase, которые позволяют очень быстро найти запись в таблице. Поэтому при создании и удалении первичного ключа создается или удаляется индекс на то поле (или поля), которое входит в первичный ключ.

Как уже было сказано, первичный ключ может содержать несколько полей. При этом будет отслеживаться уникальность сочетания значений этих полей. Например, если мы определим ключ на поля ID и NAME, то сервер будет следить за тем, чтобы во всей таблице не было одинаковых сочетаний этих полей. То есть сочетания полей ID и NAME вроде 1 и "Иванов", 2 и "Иванов" будут корректными, поскольку они отличаются значениями поля ID.

Таким образом, первичный ключ может включать несколько полей любых типов. Однако на практике самым распространенным видом ключа является счетчик – целочисленное поле, которое содержит увеличивающиеся значения. Почему так? Это является отражением давнего спора между *естественными*

и *суррогатными* ключами. Концепция естественных ключей утверждает, что в качестве ключа надо стараться использовать значения, реально существующие в предметной области, которую отражает база данных. Например, если мы разрабатываем систему учета людей для паспортного стола, то согласно этой концепции необходимо в качестве первичного ключа взять сочетание номера и серии паспорта. Действительно, каждый человек должен иметь свое уникальное сочетание номера и серии паспорта. Однако как быть с тем, что человек может поменять паспорт в течение жизни (в связи с достижением определенного возраста, при заключении брака и т. д.)? В этом случае нам будет необходимо сменить номер и серию паспорта, поставленные в соответствие конкретному человеку, т. е. фактически, сменить наш первичный ключ. Это нежелательно с точки зрения разработки приложений баз данных: при разветвленной системе связей между таблицами (этому посвящена следующая глава) может понадобиться слишком много усилий разработчика для отслеживания этой ситуации.

Поэтому в большинстве случаев используется суррогатный ключ. Суррогатный – значит искусственный, т. е. не существующий в предметной области, которую описывает наша база данных, а созданный искусственно – для удобства разработки приложений базы данных.

Как уже было сказано, обычно первичным ключом является счетчик. Некоторые СУБД, такие, как Paradox и MS SQL, имеют специальный тип – счетчик (auto increment). При добавлении в таблицу новой записи значение поля с этим типом автоматически увеличивается на величину приращения – обычно на единицы. В InterBase нет поля типа счетчик, однако есть возможность реализовать подобное поведение. Для создания поля, которое бы заполнялось автоматически при добавлении записи в таблицу, используется совокупность средств: первым из них является генератор.

Что такое *генератор*? Говоря по-простому, генератор – это именованный счетчик. Внутри базы данных мы можем создать счетчик, дать ему уникальное имя в пределах этой базы и управлять значениями этого счетчика. Это и будет генератор. Чтобы это пояснить – вот пример предложений DDL:

```
CREATE GENERATOR g1;
SET GENERATOR g1 TO 2445;
```

В этом примере в первой строчке в базе данных создается генератор с именем g1, а во второй – этому генератору присваивается значение 2445. Теперь возникает вопрос, как нам использовать полученный генератор. Чтобы получать и изменять значения генераторов, существует встроенная в InterBase функция GEN_ID. Эта функция принимает в качестве параметров имя генератора и величину приращения, которую нужно применить к данному генератору, а возвращает целочисленное значение, соответствующее значению генератора, полученному в результате прибавления к нему приращения. Вот пример вызова функции GEN_ID в тригере или хранимой процедуре:

```
Current_value = GEN_ID (g1, 1)
```

Чтобы получить значение генератора в клиентском приложении, можно воспользоваться таким запросом:

```
SELECT GEN_ID(g1, 1)FROM
RDB$ DATABASE
```

Так как в таблице RDB$ Database всегда содержится только одна запись, то мы получим в результате данного запроса значение генератора g1.

Здесь current_value – переменная (как использовать переменные в InterBase – см. в следующих главах), g1 – генератор, 1 –приращение. В этом примере в переменную current_value попадет значение генератора g1 после прибавления к нему приращения 1, т. е. следующее значение генератора!

Обратите внимание, что приращение может быть не равно единице! Более того, оно может быть даже отрицательным:

```
Current_value = GEN_ID (g1, -23)
```

В результате выполнения этой функции текущее значение генератора g1 уменьшиться на 23. Как видите, диапазон возможных применений генераторов довольно широк – его можно использовать не только для получения значений первичных ключей, но и для отслеживания глобальных изменений в базе данных.

Люди, знакомые с базами данных, могут задать вопрос: "А что будет, если одновременно несколько клиентов попробуют внести данные в одну и ту же таблицу и одновременно "дернут" генераторы? Получат ли они одно или разные значения генератора?" Однозначно, что они получат РАЗНЫЕ значения генератора. Какой бы "одновременной" ни была попытка получить значение генератора, каждый обратившийся получит свое уникальное значение. Это гарантируется самой "конструкцией" генераторов: они работают на самом низком уровне сервера и никакие процессы записи и вставки не влияют на них – часто говорят, что генераторы работают "вне контекста транзакций". Что такое транзакции, вы можете узнать в главе "Транзакции. Параметры транзакций" (ч. 1), а как устроены генераторы – в главе "Структура базы данных InterBase" (ч. 4).

Ну хорошо, в лице генераторов мы имеем надежный механизм для формирования уникальных первичных ключей. Однако как же нам воспользоваться этим механизмом? Как поместить получаемое от генератора значение в поле первичного ключа?

Для этого есть два способа – вставка первичного ключа на стороне клиента и на стороне сервера. Чтобы освоить первый способ, следует обратиться к главе "Использование основных компонентов FIBPlus", а чтобы понять второй – к главе "Триггеры" (ч. 1). Здесь мы лишь кратко скажем, в чем заключается суть обоих способов.

В случае формирования первичного ключа на клиенте происходит следующее. Когда сформирована запись, которая будет вставлена в базу данных, выполняется вызов функции GEN_ID(<имя генератора>,1) и полученное значение подставляется в сформированную запись. Происходит вставка в таблицу, при этом мы получаем гарантированно уникальный первичный ключ.

Второй способ – формирование первичного ключа на стороне сервера – вообще исключает всякую заботу на стороне клиента о том, каково будет значение первичного ключа. В этом случае при вставке записи срабатывает триггер – специальный объект базы данных, который может осуществлять какие-либо действия при вставке/удалении/изменении записей в таблицах. И в этом триггере

происходит вызов функции GEN_ID, получение нужного значения генератора и вставка его в таблицу.

Достоинством второго способа является то, что при разработке клиентского приложения совершенно не надо заботиться о формировании первичного ключа, достаточно лишь раз написать нужный триггер. Но его недостатком является то, что мы не можем получить в приложении значение сформированного ключа сразу после вставки! При использовании первого способа мы, хотя и сами должны каждый раз при вставке первичного ключа заботиться о его формировании, можем получить его значение. Какой способ лучше – однозначно сказать нельзя, все зависит от конкретной задачи, но возможные варианты разрешения вопросов работы с первичным ключом будут еще не раз затронуты далее в этой книге.

Заключение

Итак, в этой главе мы рассмотрели, как создавать и модифицировать таблицы в InterBase, а также как обращаться с первичными ключами. Таким образом, мы рассмотрели главные объекты в InterBase, которые можно условно назвать статическими, поскольку они только хранят информацию и не осуществляют ее преобразования. Далее мы поведем разговор о методах контроля за информацией и о преобразовании информации внутри базы данных.

Индексы

Концепция, положенная в основу индексов, проста и наглядна и является одной из важнейших основ проектирования баз данных. На основе индексов базируются многие основополагающие объекты базы данных, к тому же правильное использование индексов является ключом к улучшению производительности приложений баз данных.

Что же представляет собой индекс? Индекс – это упорядоченный указатель на записи в таблице. *Указатель* означает, что индекс содержит значения одного или нескольких полей в таблице и адреса страниц данных, на которых располагаются эти значения (про страницы данных см. главу "Структура базы данных InterBase") (ч. 4). Другими словами, индекс состоит из пар значений "значение поля" – "физическое расположение этого поля". Таким образом, по значению поля (или полей), входящего в индекс, при помощи индекса можно быстро найти то место в таблице, где располагается запись, содержащая это значение.

Упорядоченный – означает, что значения полей, хранящихся в индексе, упорядочены.

Очень часто индекс сравнивают с библиотечным каталогом, в котором все книги записаны на карточки и упорядочены каким-то образом: по алфавиту или по темам, а в каждой карточке написано, где именно в хранилище располагается данная книга.

Для чего нужны индексы?

Единственное, чему способствуют индексы, – это ускорению поиска записи по ее индексированному полю (индексированное – значит входящее в индекс).

Итак, основная функция индексов – обеспечивать быстрый поиск записи в таблице. Любое использование индексов сводится именно к этому.

Как реализована эта функция поиска? На входе функции мы имеем значение индексированного поля (или нескольких полей). В результате поиска мы должны получить всю запись, в которой индексированное поле имеет заданное значение. Сначала в индексе (точнее, в упорядоченном массиве значений индексированного поля) ищется нужное значение, затем берется адрес страницы данных, на которой лежит искомая запись, сервер перемещается на эту страницу и читает найденную запись. Выглядит довольно громоздко, однако поиск с помощью индекса происходит во много раз быстрее, чем при последовательном переборе всех значений из таблицы.

Если продолжить аналогию индекса с библиотечным каталогом, то поиск записи с помощью индекса очень похож на поиск книги с помощью карточки. Стоит нам найти книгу в относительно небольшом по объему каталоге (по сравнению со всем библиотечным хранилищем), как сразу получаем информацию о точном местонахождении книги и можем направиться прямиком туда. Поиск же без использования индекса можно сравнить с последовательным перебором всех книг в библиотеке!

Перебор всех записей в таблице называется *прямым* или *естественным* (NATURAL). Надо сказать, что, несмотря на мощности современных компьютеров, при достаточно большом количестве записей в таблице естественный перебор может быть очень долгим процессом.

Как устроены индексы

Индекс не является частью таблицы – это отдельный объект, связанный с таблицей и другими объектами базы данных. Это очень важный момент реализации СУБД, который позволяет отделить хранение информации от ее представления.

InterBase, как и всякая другая реляционная база данных, хранит записи в таблицах в неупорядоченном виде, т. е. совершенно не заботится о том, как физически располагаются записи в таблице. Неупорядоченность хранения означает, что две записи, добавляемые в таблицу одна за другой, совсем не обязательно окажутся "рядом". Более того, данные, извлекаемые из таблицы, также не имеют какого-либо порядка, кроме того, который явно должен быть указан пользователем, составляющим запрос на выборку.

Однако без упорядочения хранящихся данных обойтись невозможно: конечные пользователи приложений хотят видеть свои данные в определенном порядке – например, фамилии людей по алфавиту. Задачу представления данных в упорядоченном виде решают индексы. Значения полей, входящих в индекс упорядочены и представлены в особом виде, оптимизированном для поиска нужных значений (а именно это и нужно для построения упорядоченных последовательностей). Отделение хранения данных от их представления дает дополнительные преимущества по сравнению с непосредственной сортировкой – исходную таблицу может потребоваться отсортировать по-разному. Тогда на помощь приходят индексы – их может быть до 64 на каждую таблицу!

Если говорить о реализации индексов на физическом уровне, то они представляют двоичное дерево, узлы которого представляют собой пары "значение поля в индексе" – "расположение данных в таблице". Поиск нужной записи в индексе идет с помощью механизма хеш-поиска – одного из самых быстрых алгоритмов поиска.

Применение индексов

Теперь, когда ясно, что можно требовать от индексов, настало время разобраться с тем, какую роль они играют в базе данных. Индексы используются в трех основных случаях:

1. Ускорение выполнения запросов. Индексы создаются для полей, которые используются в условиях поиска SQL-запросов.
2. Обеспечение уникальности значений в полях; Ограничение первичного ключа (о которых рассказывалось в главе "Таблицы. Первичные ключи") требует, чтобы во всей таблице не нашлось двух одинаковых значений полей, входящих в первичный ключ. Чтобы выполнить это условие, необходимо при каждой вставке новой записи производить поиск такого же значения, которые будет вставлено. Для поиска записи используется особая разновидность индекса – уникальный индекс (см. ниже).
3. Обеспечение ссылочной целостности. Ограничения внешних ключей Foreign key (которые рассмотрены в главе "Ограничения базы данных") используются для проверки того, чтобы вставляемые в таблицу значения обязательно

существовали в другой таблице. При создании внешнего ключа автоматически создается индекс, который применяется как для ускорения запросов, использующих соединение таблиц, так и для проверки условий внешнего ключа.

Вот вкратце и все возможные применения индексов. Теперь мы рассмотрим особенности каждого случая более подробно и ответим на ряд часто возникающих вопросов, связанных с применением индексов.

Ускорение выполнения запросов с помощью индексов

Выше описано, что применение индексов может значительно ускорить выполнение запросов. Это действительно так для большинства случаев, но есть и определенные оговорки. Сначала ответим на вопрос, часто возникающий у тех, кто познакомился с индексами. Раз индексы ускоряют выборку из базы данных, почему бы не проиндексировать все поля в таблице? Есть два момента, препятствующих всеобщей индексации, – это дисковое пространство и издержки при модификации данных в таблице. Каждый создаваемый индекс имеет объем, равный объему данных в индексированном поле, плюс объем данных о расположении записей. Если создать индексы на каждое поле в таблице, то их суммарный объем будет больше, чем объем данных в таблице! Поэтому создание большого количества индексов приводит к большому расходу дискового пространства.

Второй момент более важен – это издержки при модификации данных в таблице. В реляционной СУБД, как вы знаете, записи в таблицах неупорядочены и потому добавление/удаление записей происходят без значительных затрат ресурсов сервера. Даже если удаляется запись из середины базы данных, то не происходит перемещения объемов данных для того, чтобы закрыть "дыру", – это попросту не нужно: сервер просто пометит освободившееся место и при случае запишет туда что-нибудь. Что касается добавления, то оно почти всегда происходит в конец таблицы. Однако хотя основные данные в таблице и не "дергаются" сервером при модификации, но данные, хранящиеся в индексах, переупорядочиваются каждый раз при добавлении/удалении записей! То есть серверу при добавлении записи в середину таблицы, например, приходится перестраивать индекс! Конечно, реализация индекса некоторым образом рассчитана на частые перестройки, но эти действия все же занимают время и ресурсы процессора и при слишком большом количестве индексов в таблице модификация данных в ней может быть в десятки раз медленнее, чем у такой же таблицы без индексов!

Это две основные причины, которые препятствуют всеобщей индексации. Помимо них есть и еще несколько замечаний, ограничивающих применение индекса. Первое – это правило 20 %. Оно гласит, что если запрос на выборку возвращает более 20 % записей из таблицы, то использование индекса может замедлить выборку данных! Конечно, ситуация зависит от конкретного запроса и условий, наложенных на выборку, но нужно помнить, что 20 % записей являются порогом, когда эффективность использования индексов ставится под вопрос. Второе замечание формулируется не так очевидно. Оно связано с работой оптимизатора InterBase.

Оптимизатор – это совокупность механизмов, которые разрабатывают *план* выполнения запроса. Когда пользователь передает InterBase какой-либо

SQL-запрос, он указывает, ЧТО должен вернуть сервер в результате выполнения запроса, но не определяет, КАК сервер должен выполнять запрос. Оптимизатор на основе переданного запроса строит план его выполнения, т. е. откуда и в каком порядке будут браться данные для выполнения запроса, какие индексы будут при этом использоваться. Когда сервер анализирует условия на выборку (это в основном части выражения WHERE, ORDER BY и т. д.), то для каждого поля, входящего в условие, сервер пытается использовать индекс. К сожалению, алгоритм создания плана несовершенен и оптимизатор часто использует индексы, которые не слишком эффективны для конкретного запроса, из-за чего может существенно замедлиться время выполнения. Поэтому создание лишних индексов может привести к созданию неоптимальных планов. Конечно, разработчики серверов борются с таким положением вещей, и последние версии InterBase и Firebird все лучше и лучше справляются с оптимизацией.

Третьим случаем, когда индекс не нужен, являются поля с ограниченным набором значений – например, поле, которое хранит информацию о поле человека и содержит только два возможных значения – "м" и "ж"; нет никакого смысла индексировать это поле.

Итак, основные ограничения на создание индексов мы рассмотрели. Теперь следует рассмотреть вопрос, когда следует использовать индексы, чтобы добиться улучшения производительности. Существует 3 основных случая, когда необходимо проиндексировать поле:

1. Когда это поле используется в условиях поиска в запросах.
2. Когда соединения таблиц (JOIN) используют это поле.
3. Когда это поле используется в предложениях сортировки ORDER BY.

Если поле применяется указанным выше образом, то создание индекса на него может привести к улучшению производительности запроса.

Давайте рассмотрим синтаксис создания индексов. Вот полный формат команды DDL, который позволяет создавать индексы:

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

Минимальным выражением, создающим индекс, является следующее:

```
CREATE INDEX my_index ON Table_example(ID)
```

В этом примере создается индекс с именем my_index для таблицы Table_example, причем индексированным полем является поле ID. Индекс является возрастающим, т. е. значения в нем упорядочены по возрастанию, а также неуникальным, т. е. значит, что поле ID может иметь несколько одинаковых значений. Это, конечно же, самый простой пример индекса – самый распространенный.

Как видно из описания синтаксиса, индекс может содержать не одно, а несколько полей. Такой индекс используется при часто выполняющихся запросах, которые содержат в условиях поиска или сортировки сочетание индексированных полей. Например, если у нас есть таблица, содержащая поля Фамилия, Имя, Отчество, то при запросе, использующем сортировку по ФИО, будет применен такой индекс. Вообще говоря, необязательно вводить условия на все 3 поля,

применяемые в индексе, чтобы использовать его преимущества. Если мы желаем сортировать результат запроса, то индекс будет использован в случае, если первое поле в условии сортировки совпадает с первым полем в индексе, например наш индекс будет задействован в случае сортировки по Фамилии и Имени.

В документации для оптимизации выполнения запроса, содержащего в предложении WHERE соединение полей с условием OR рекомендуется, использовать не составной индекс, а несколько одинарных индексов на все поля, входящие в условие OR.

К вопросу о порядке сортировки индекса: как видно, он может быть либо возрастающим (ASC[ENDING]), либо убывающим (DESC[ENDING]). Зачем нужны разные порядки сортировки? Очевидно, для разных сортировок! Если мы желаем сортировать людей по фамилии в возрастающем порядке, то создаем возрастающий индекс (ASC), а если в убывающем (от Я до А) – то убывающий! А если хотим и то и другое, то необходимо создавать оба индекса.

Обеспечение ссылочной целостности с помощью индексов

В определении индекса имеется еще одна опция – UNIQUE. Если ее указать, то индекс позволит заносить в таблицу только уникальные значения. Фактически это служит основой для реализации уникальных ключей (UNIQUIE KEY). Уникальные ключи широко используются в базах данных. То есть PK – это уникальный ключ-индекс, но не всякий UK – это PK. Выше речь шла только о PK. Первичный ключ (Primary key) – самый распространенный вид уникального ключа. При создании первичного ключа на таблицу автоматически создается уникальный индекс, который получает имя, составленное из RDB$PRIMARYNNN, где NNN – последовательный уникальный в пределах базы данных номер. Таким образом, с помощью уникального индекса реализуются два из важнейших ограничений ссылочной целостности – уникальный ключ и первичный ключ. Очевидно, что понятие уникальности несовместимо с понятием неопределенного значения, т. е. другими словами, в полях, содержащихся в уникальных индексах, не должно быть значений типа NULL. Перед созданием уникального индекса на поле следует придать ему статус NOT NULL. Если индекс создается для уже существующих данных, то при создании будет проверено, не содержит ли индексированное поле повторяющихся значений. И если содержит, то в создании индекса будет отказано.

Помимо ограничений уникального и первичного ключа, механизм индексов лежит в основе реализации еще одного ограничения ссылочной целостности – внешнего ключа. Ограничение внешнего ключа накладывается на одно или несколько полей какой-либо таблицы и препятствует внесению в эти поля таких значений, которые не входят в первичный ключ другой, родительской таблицы. Для реализации внешнего ключа, т. е. для осуществления проверки того, существует ли значение в родительской таблице, автоматически создается особый индекс. Он имеет наименование RDB$FOREIGNNN, где NNN – последовательный уникальный в пределах базы данных номер.

Почему именно механизм индексов используется для реализации ограничений ссылочной целостности? Дело в том, что индексы в InterBase находятся

в особом, привилегированном положении – говорят, что они выполняются вне контекста транзакций. Это очень важное свойство. О транзакциях мы поговорим позже, в посвященной им главе, а пока лишь скажем, что нахождение индексов вне транзакций означает, что все пользователи, одновременно работающие с данными в одной и той же таблице, вынуждены соблюдать ограничения ссылочной целостности.

Оптимизация производительности индексов

В названии этого раздела можно обнаружить некоторый парадокс – индексы, как говорилось выше, служат для того, чтобы ускорить выполнение запросов, и оказывается, что их самих надо тоже оптимизировать! Но что делать (такова жизнь) – кто-то должен заботиться и об индексах.

Что же случается с индексами? Почему они "теряют форму"? Нам придется еще раз сказать о том, что индексы реализованы в виде двоичного дерева. И когда в таблицу добавляется (изменяется, удаляется – выберите по вкусу) новая запись, в дерево добавляется новая веточка. Причем веточки добавляются не в середину дерева, а на концах других веточек. Постепенно дерево становится все более "раскидистым" (также говорят – несбалансированным), а поиск по нему – все менее эффективным. Поправить положение может перестройка дерева или (в некоторых случаях) пересчет статистики. Периодически требуется пересоздавать индекс, чтобы восстанавливать его производительность. Пересоздание индекса происходит в следующих случаях:

1. При перестройке индекса с помощью команды ALTER INDEX.
2. При удалении и повторном создании индекса командами DROP INDEX и CREATE INDEX.
3. При резервном копировании и восстановлении из резервной копии с использованием инструмента gbak.

Также можно использовать пересчет статистики. Но надо понимать, что это действие не изменяет состояние индекса, а просто сообщает оптимизатору точные данные о его состоянии, что позволяет правильно использовать этот индекс. Другими словами, пересчет статистики – это не "лечение" индекса, а только точная диагностика его состояния.

Рассмотрим подробнее все эти способы оптимизации индексов.

Использование команды ALTER INDEX имеет следующий формат:

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

Здесь name – имя индекса, а ACTIVE и INACTIVE – два состояния индекса, в которые его можно перевести при помощи команды ALTER INDEX. Параметр ACTIVE означает, что индекс активен и может применяться во всех запросах и процедурах. Установка индекса в INACTIVE (неактивен) приводит к отключению его использования. Для перестройки дерева надо последовательно выполнить две команды:

```
ALTER INDEX name INACTIVE;
ALTER INDEX name ACTIVE;
```

При этом индекс будет перестроен. Использование ALTER INDEX имеет ряд ограничений: с его помощью нельзя перестроить индексы, используемые в первичных, уникальных и внешних ключах; нельзя перестроить индекс, если он используется в данный момент каким-либо запросом; а также для изменения индекса необходимо иметь права администратора (SYSDBA) или быть создателем данного индекса.

Пересоздание индекса с помощью команд DROP INDEX и CREATE INDEX приводит к полному удалению индекса из базы данных, а затем к его созданию с чистого листа. Синтаксис команды DROP INDEX очевиден:

```
DROP INDEX имя_индекса;
```

После удаления необходимо создать индекс с тем же именем и параметрами с помощью команды CREATE INDEX, синтаксис которой мы уже рассматривали.

У способа перестройки индекса путем его полного пересоздания есть ограничения, аналогичные ограничениям на использование ALTER INDEX.

Третий способ перестройки индекса основан на свойстве резервных копий баз данных InterBase, которые создаются с помощью утилиты gbak. Дело в том, что при резервном копировании данные, входящие в индекс, не сохраняются в резервной копии, а хранится только определение индекса. При восстановлении из резервной копии индекс создается заново. Подробнее о резервном копировании см. главу "Резервное копирование и восстановление из резервной копии" (ч. 4).

Четвертый способ улучшить производительность индекса – это собрать статистику по индексам с помощью команды SET STATISTICS. Статистика таблицы – это величина в пределах от 0 до 1, значение которой зависит от числа различных (неодинаковых) записей в таблице. Оптимизатор InterBase использует статистику для определения эффективности применения того или иного индекса в запросе. Когда число записей в таблице может сильно изменяться (например, при большом количестве вставок или удалений), то пересчет статистики может значительно улучшить производительность.

Команда пересчета статистики следующая:

```
SET STATISTICS INDEX name;
```

Здесь name – имя индекса, для которого пересчитывается статистика.

Пересчет статистики не перестраивает индекс и потому свободен от большинства ограничений, накладываемых на описанные выше способы улучшения производительности, за исключением того, что пересчитывать статистику может либо создатель индекса, либо системный администратор (пользователь с именем SYSDBA). Правильная статистика дает оптимизатору возможность принять верное решение об использовании или неиспользовании какого-либо индекса.

Мы рассмотрели несколько способов улучшить производительность индексов. С помощью команд ALTER INDEX и DROP/CREATE INDEX можно перестраивать любые индексы, за исключением системных, создаваемых автоматически индексов, служащих для поддержания ссылочной целостности. Чтобы перестроить эти индексы, необходимо воспользоваться командами изменения и создания таблиц – ALTER TABLE и CREATE TABLE, так как эти индексы являются неотъемлемой частью табличных ключей.

Ограничения базы данных

Эта глава посвящена ограничениям базы данных InterBase. Ограничения базы данных, – это правила, которые определяют взаимосвязи между таблицами и могут проверять и изменять данные в базе данных. Реализованы эти правила в виде особых объектов базы данных.

Главное преимущество использования ограничений состоит в возможности реализовать проверку данных, а значит, и часть бизнес-логики приложения на уровне базы данных, т. е. централизовать и упростить ее, а значит, сделать разработку приложений баз данных проще и надежнее.

Часто начинающие разработчики пренебрегают использованием ограничений базы данных, считая, что они стесняют возможность творчества. Однако на самом деле такое мнение происходит от недостаточного знания теории и практики проектирования баз данных.

В то же время наиболее опытные разработчики позволяют себе отказаться от использования некоторых видов ограничений, за счет чего их приложения выигрывают в быстродействии. Опыт высококвалифицированных разработчиков позволяет им очень хорошо понимать работу сервера и точно предсказывать его поведение в сложных случаях, поэтому начинающим программистам InterBase лучше не апеллировать к подобным действиям опытных коллег.

В рамках данной книги мы не рассматриваем проектирование баз данных, поэтому для получения дополнительной информации по этому вопросу следует обратиться к списку литературы в конце книги. Здесь же мы лишь проведем обзор всех видов ограничений в базе данных InterBase и рассмотрим примеры их применения.

Виды ограничений в базе данных

Существуют следующие виды ограничений в базе данных InterBase:

- первичный ключ – PRIMARY KEY;
- уникальный ключ – UNIQUE KEY;
- внешний ключ – FOREIGN KEY

  – может включать автоматические триггеры ON UPDATE и ON DELETE;
- проверки – CHECK.

В предыдущих главах уже упоминались некоторые из этих ограничений, что было необходимо для логичного изложения материала, но теперь мы рассмотрим их синтаксис, применение и реализацию более обстоятельно.

Ограничения базы данных бывают двух типов – на основе одного поля и на основе нескольких полей таблицы. Синтаксис обоих видов ограничений приведен ниже.

```
<col_constraint> = [CONSTRAINT constraint] <constraint_def>
[ <col_constraint> ...]
<constraint_def> = {UNIQUE | PRIMARY KEY
| CHECK ( <search_condition>)
```

```
| REFERENCES other_table [( other_col [, other_col …])]
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
}
```

Для ограничений на основе нескольких полей синтаксис следующий:

```
<tconstraint> = [CONSTRAINT constraint] <tconstraint_def>
[< tconstraint> ...]
<tconstraint_def> = {{PRIMARY KEY | UNIQUE} ( col [, col …])
| FOREIGN KEY ( col [, col …]) REFERENCES other_table [( oth-
er_col [, other_col …])]
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
| CHECK ( <search_condition>)}
```

Разница в синтаксисе между ограничениями на основе одного поля и на основе нескольких очевидна – в последних можно указать несколько полей, которые входят в ограничение. В случае ограничения на основе одного поля все описанные опции относятся только к текущему полю.

Конечно, у этих двух типов ограничений отличается способ их применения: ограничения на основе одного поля просто дописываются к определению нужного поля, а ограничения на основе нескольких полей указываются через запятую в общем определении таблицы. Подробные примеры приведены в следующих разделах этой главы.

Пример типичного ограничения

Фактически ограничения на основе одного поля являются частным случаем ограничений на основе нескольких полей.

Пример создания ограничения первичного ключа с использованием этих двух различных подходов приведен ниже. Давайте создадим таблицу, содержащую только одно поле и наложим на нее ограничение первичного ключа.

Первичный ключ с использованием синтаксиса ограничения на основе одного поля:

```
CREATE TABLE test1(
ID_PK  INTEGER CONSTRAINT pktest NOT NULL PRIMARY KEY);
```

В этом примере создается первичный ключ с именем pktest на поле ID_PK. Получаем весьма компактное описание в одну строчку.

Для той же самой цели можно воспользоваться синтаксисом ограничений на основе нескольких полей:

```
CREATE TABLE test2(
ID_PK  INTEGER NOT NULL,
CONSTRAINT pktst PRIMARY KEY (ID_PK));
```

Создание ограничений

Давайте рассмотрим создание ограничений подробнее. Первой в описании общего синтаксиса ограничений идет опция [CONSTRAINT constraint]. Как видите, эта опция взята в квадратные скобки и, значит, необязательна.

С помощью этой опции можно задавать имя создаваемому ограничению и в случае использования синтаксиса ограничений на основе одного поля, и в случае ограничений на основе нескольких полей.

Если не указать имя для ограничения, InterBase автоматически сгенерирует его. Но лучше все же явно назначить имя создаваемому ограничению, чтобы улучшить читабельность схемы базы данных, а также упростить управление ограничениями в дальнейшем.

Назначив имя ограничению, необходимо определить его тип. Рассмотрим различные типы ограничений в том порядке, как они указаны в описании общего синтаксиса ограничений.

Первичный и уникальный ключи

Первичные ключи являются одним из основных видов ограничений в базе данных. Они применяются для однозначной идентификации записей в таблице. Допустим, мы храним в базе данных список людей. Вполне вероятно, что могут появиться два (или больше) человека с одинаковыми фамилией, именем и отчеством. Как же гарантированно отличить одного человека от другого (конечно, речь идет о том, чтобы отличить одного человека от другого на основании информации, хранящейся в базе данных)?

В данном случае "человек" представлен одной записью в таблице, поэтому можно задаться более общим вопросом – как отличить одну запись в (любой) таблице от другой записи в этой же таблице. Для этого используются ограничения – *первичные ключи*. Первичный ключ представляет собой одно или несколько полей в таблице, сочетание которых уникально для каждой записи. Для одной таблицы не существует повторяющихся значений первичного ключа.

Уникальные ключи несут аналогичную нагрузку – они также служат для однозначной идентификации записей в таблице. Отличие первичных ключей от уникальных состоит в том, что первичный ключ может быть в таблице только один, а уникальных ключей – несколько. Надо отметить, что и первичный и уникальный ключ могут быть использованы в качестве ссылочной основы для внешних ключей (см. далее).

> Формальное описание понятий первичного и уникального ключей, а также других важных определений можно найти в приложении "Глоссарий" в конце книги.

Синтаксис создания первичного и уникального ключа на основе единственного поля следующий:

```
<pkukconstraint> = [CONSTRAINT constraint] {PRIMARY KEY |
UNIQUE}
```

Примеры первичных и уникальных ключей:

```
CREATE TABLE pkuk(
pk NUMERIC(15,0) NOT NULL PRIMARY KEY, /*первичный ключ*/
```

```
uk1 VARCHAR(50) NOT NULL UNIQUE,/*уникальный ключ */
uk2 INTEGER NOT NULL UNIQUE /* еще уникальный ключ */);
```

Синтаксис создания первичного и уникального ключей на основе нескольких полей:

```
<pkuktconstraint> = [CONSTRAINT constraint] {PRIMARY KEY |
UNIQUE} ( col [, col …])
```

Такой синтаксис позволяет создавать ключи на основе комбинации полей. Вот примеры создания первичных и уникальных ключей из нескольких полей:

```
CREATE TABLE pkuk2(
Number1 INTEGER NOT NULL,
Name1 VARCHAR(50) NOT NULL,
Kol INTEGER NOT NULL,
Stoim NUMERIC(15,4) NOT NULL,
CONSTRAINT pkt PRIMARY KEY (Number1, Name1), /*первичный ключ pkt на
основе двух полей*/
CONSTRAINT ukt1 UNIQUE (kol, Stoim)); /*уникальный ключ ukt1 на основе
двух полей*/
```

Обратите внимание, что все поля, входящие в состав первичного и уникального ключей, должны быть объявлены как NOT NULL, так как эти ключи не могут иметь неопределенного значения.

Помимо создания ограничения первичных и уникальных ключей в момент создания таблицы имеется возможность добавлять ограничения в уже существующую таблицу. Для этого используется предложение DDL: ALTER TABLE. Синтаксис добавления ограничений первичного или уникального ключа в существующую таблицу аналогичен описанному выше:

```
ALTER TABLE tablename
ADD [CONSTRAINT constraint] {PRIMARY KEY | UNIQUE} ( col [, col …])
```

Давайте рассмотрим пример создания первичного и уникального ключа с помощью ALTER TABLE. Сначала создаем таблицу:

```
CREATE TABLE pkalter(
ID1 INTEGER NOT NULL,
ID2 INTEGER NOT NULL,
UID VARCHAR(24));
```

Затем добавляем ключи. Сначала первичный:

```
ALTER TABLE pkalter
ADD CONSTRAINT pkal1 PRIMARY KEY (id1, id2);
```

Затем уникальный ключ:

```
ALTER TABLE pkalter
ADD CONSTRAINT ukal UNIQUE (uid);
```

Важно отметить, что добавление (а также удаление) ограничений первичных и уникальных ключей к таблице может производить только владелец этой таблицы или системный администратор SYSDBA (подробнее о владельцах и пользователе SYSDBA см. главу "Безопасность в InterBase: пользователи, роли и права") (ч. 4).

Внешние ключи

Следующим ограничением, которое часто используется в базах данных InterBase, является ограничение внешнего ключа. Это очень мощное средство для поддержания ссылочной целостности в базе данных, которое позволяет не только контролировать наличие правильных ссылок в базе данных, но и автоматически управлять этими ссылками!

Смысл создания внешнего ключа следующий: если две таблицы служат для хранения взаимосвязанной информации, то необходимо гарантировать, чтобы эта взаимосвязь была всегда корректной. Пример – документ "накладная", содержащий общий заголовок (дата, номер накладной и т. д.) и множество подробных записей (наименование товара, количество и т. д.).

Для хранения такого документа в базе данных создается две таблицы – одна для хранения заголовков накладных, а вторая – для хранения содержимого накладной – записей о товарах и их количестве. Такие таблицы называются *главной* и *подчиненной* или *таблицей-мастером* и *деталь-таблицей*.

Согласно здравому смыслу невозможно существование содержимого накладной без наличия ее заголовка. Другими словами, мы не можем вставлять записи о товарах, не создав заголовок накладной, а также не можем удалять запись заголовка, если существуют записи о товарах.

Для реализации такого поведения таблица заголовка соединяется с таблицей подробностей с помощью ограничения внешнего ключа.

Давайте рассмотрим смысл наложения ограничений внешнего ключа на примере таблиц, содержащих информацию о накладных.

Для этого создадим две таблицы для хранения накладной – таблицу TITLE для хранения заголовка и таблицу INVENTORY для хранения информации о товарах, входящих в накладную.

```
CREATE TABLE TITLE(
ID_TITLE INTEGER NOT NULL Primary Key,
DateNakl DATE,
NumNakl INTEGER,
NoteNakl VARCHAR(255));
```

Обратите внимание на то, что мы сразу определили первичный ключ в таблице заголовка на основе поля ID_TITLE. Остальные поля таблицы TITLE содержат тривиальную информацию о заголовке накладной – дату, номер, примечание.

Теперь определим таблицу для хранения информации о товарах, входящих в накладную:

```
CREATE TABLE INVENTORY(
ID_INVENTORY INTEGER NOT NULL PRIMARY KEY,
FK_TITLE INTEGER NOT NULL,
ProductName VARCHAR (255),
Kolvo DOUBLE PRECISION,
Positio INTEGER);
```

Давайте рассмотрим, какие поля входят в таблицу INVENTORY. Во-первых, это ID_INVENTORY – первичный ключ этой таблицы. Затем идет целочисленное поле FK_TITLE, которое служит для ссылки на идентификатор заголовка

ID_TITLE в таблице заголовков накладных. Далее идут поля ProductName, Kolvo и Positio, описывающие наименование товара, его количество и позицию в накладной.

Для нашего примера важнее всего поле FK_TITLE. Если мы захотим вывести информацию о товарах определенной накладной, то нам следует воспользоваться следующим запросом, в котором параметр mas_ID_TITLE определяет идентификатор заголовка:

```
SELECT *
FROM INVENTORY I1
WHERE I1.FK_TITLE=?mas_ID_TITLE
```

В сущности, в описываемой ситуации ничто не мешает заполнить таблицу INVENTORY записями, ссылающимися на несуществующие записи в таблице TITLE. Также ничего не препятствует удалению заголовка уже существующей накладной, в результате чего записи о товарах могут стать "бесхозными".

Сервер не будет препятствовать всем этим вставкам и удалениям. Таким образом, контроль за целостностью данных в базе данных полностью возлагается на клиентское приложение. А ведь с одной базой данных могут работать несколько приложений, разрабатываемых, быть может, разными программистами, что может привести к различной интерпретации данных и к ошибкам.

Поэтому необходимо явно наложить ограничение на то, что в таблицу INVENTORY могут помещаться лишь такие записи о товарах, которые имеют корректную ссылку на заголовок накладной. Собственно это и есть ограничение *внешнего ключа*, которое позволяет вставлять в поля, входящие в ограничения, только те значения, которые есть в другой таблице.

Такое ограничение можно организовать с помощью внешнего ключа. Для данного примера необходимо наложить ограничения внешнего ключа на поле FK_TITLE и связать его с первичным ключом ID_TITLE в TITLE. Добавить внешний ключ в уже существующую таблицу можно следующей командой:

```
ALTER TABLE INVENTORY
ADD CONSTRAINT fktitle1 FOREIGN KEY(FK_TITLE) REFERENCES TI-
TLE(ID_TITLE)
```

> Часто при добавлении внешнего ключа возникает ошибка object is in use (объект используется). Дело в то, что для создания внешнего ключа, необходимо открыть базу данных в монопольном режиме – чтобы одновременно не было других пользователей. Также нельзя производить никаких обращений к модифицируемой таблице – это может вызвать object is in use.

Здесь INVENTORY – имя таблицы, на которую накладывается ограничение внешнего ключа; fktitle1 – имя внешнего ключа; FK_TITLE – поля, составляющие внешний ключ; TITLE – имя таблицы, предоставляющей значения (ссылочную основу) для внешнего ключа; ID_TITLE – поля первичного или уникального ключа в таблице TITLE которые являются ссылочной основой для внешнего ключа.

Полный синтаксис ограничения внешнего ключа (с возможностью создавать ограничения на основании нескольких полей) приведен ниже:

```
<tconstraint> = [CONSTRAINT constraint] FOREIGN KEY ( col [,
col …]) REFERENCES other_table[( other_col [, other_col …])]
 [ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
```

Как видите, определения содержат большой набор опций. Для начала давайте рассмотрим базовое определение внешнего ключа, которое наиболее часто используется в реальных базах данных, а затем разберем возможные опции.

Чаще всего употребляются декларативная форма ограничения внешнего ключа, когда указывается набор полей (col [, col …]), которые будут составлять ограничение; таблица other_table, которая содержит в полях [(other_col [, other_col …]) список возможных значений для внешнего ключа.

Пример такого определения при создании таблицы:

```
CREATE TABLE Inventory2(
…
FK_TABLE INTEGER NOT NULL CONSTRAINT fkinv REFERENCES
TITLE(ID_TITLE)
…);
```

Обратите внимание, что в этом определении опущены ключевые слова FOREIGN KEY, а также подразумевается, что в качестве внешнего ключа будет использоваться единственное поле – FK_TITLE.

А в следующем примере приведена более полная форма создания внешнего ключа одновременно с таблицей:

```
CREATE TABLE Inventory2(
…
FK_TABLE INTEGER NOT NULL,
CONSTRAINT fkinv FOREIGN KEY (FK_TABLE) REFERENCES
TITLE(ID_TITLE)
…);
```

Использование NULL в полях внешнего ключа

В полях, на основе которых создается внешний ключ, допускается применение NULL-полей. Эта возможность добавлена для разрешения взаимных ссылок. Например, если есть две таблицы, ссылающиеся друг на друга с помощью внешних ключей. Если не разрешить пустую ссылку (т. е. на NULL) в этих внешних ключах, то в связанные таблицы невозможно будет добавить ни одной записи: чтобы добавить запись в первую таблицу, надо будет иметь запись во второй таблице, и наоборот.

Использование NULL в качестве пустой ссылки позволяет организовать взаимные ссылки двух перекрестно ссылающихся таблиц, а также хранить иерархические структуры в реляционных таблицах – при этом корневые узлы ссылаются на "пустые" записи (т. е. просто содержат NULL).

Расширенные возможности поддержки ссылочной целостности с помощью внешнего ключа

Обычно вполне достаточно декларативного варианта ограничения внешнего ключа, при котором сервер только следит за тем, чтобы в таблицу с внешним ключом нельзя было вставить некорректные значения или – при попытке сделать это возникает ошибка. Но InterBase позволяет выполнять ряд автоматических действий при изменении/удалении внешнего ключа. Для этого служит следующий набор опций внешнего ключа:

```
[ON DELETE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
[ON UPDATE {NO ACTION|CASCADE|SET DEFAULT|SET NULL}]
```

Эти опции позволяют определить различные действия при изменении или удалении значения внешнего ключа.

Например, мы можем указать, что при удалении первичного ключа в таблице-мастере необходимо удалять все записи с таким же внешним ключом в подчиненной таблице. Для этого следует так определить внешний ключ:

```
ALTER TABLE INVENTORY
ADD CONSTRAINT fkautodel
FOREIGN KEY (FK_TITLE) REFERENCES TITLE(ID_TITLE)
ON DELETE CASCADE
```

Фактически для реализации этих действий создается системный триггер, который и выполняет определенные действия. В табл. 1.2 приведено описание происходящих действий при различных опциях (обратите внимание, что опции NO ACTION|CASCADE|SET DEFAULT|SET NULL не могут использоваться совместно в одном предложении ON XXX).

Таблица 1.2

Событие	Действие			
	NO ACTION	CASCADE	SET DEFAULT	SET NULL
ON DELETE	При удалении внешнего ключа ничего не делать – используется по умолчанию	При удалении удалить все связанные записи из подчиненной таблицы	При изменении установить поле внешнего ключа в значение по умолчанию	При изменении установить поле внешнего ключа в NULL
ON UPDATE	При изменении ничего не делать – используется по умолчанию	При изменении записи изменить во всех связанных записях в подчиненных таблицах	При удалении установить поле внешнего ключа в значение по умолчанию	При удалении установить поле внешнего ключа в NULL

Если мы ничего не указываем или указываем NO ACTION, то необходимо позаботиться об изменении внешнего ключа (в случае изменения первичного) самостоятельно, а при удалении первичного ключа предварительно удалить записи из подчиненной таблицы.

Осторожно надо обращаться с опцией CASCADE: неосторожное ее использование может привести к удалению большого количества связанных записей.

Ограничение CHECK

Одним из наиболее полезных ограничений в базе данных является ограничение проверки. Идея его очень проста – проверять вставляемое в таблицу значение на какое-либо условие и, в зависимости от выполнения условия, вставлять или не вставлять данные.

Синтаксис его достаточно прост:

```
<tconstraint> = [CONSTRAINT constraint] CHECK (
<search_condition>)}
```

Здесь constraint – имя ограничения; <search_condition> – условие поиска, в котором в качестве параметра может участвовать вставляемое/изменяемое значение. Если условие поиска выполняется, то вставка/изменение этого значения разрешаются, если нет – возникает ошибка.

Самый простой пример проверки:

```
create table checktst(
ID integer CHECK(ID>0));
```

Эта проверка устанавливает, больше ли нуля вставляемое/изменяемое значение поля ID, и в зависимости от результата позволяет вставить/изменить новое значение или возбудить исключение (см. главу "Расширенные возможности языка хранимых процедур InterBase" (ч. 1)).

Возможны и более сложные варианты проверок. Полный синтаксис условия поиска <search_condition> следующий:

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

Таким образом, CHECK предоставляет большой набор опций для проверки вставляемых/изменяемых значений. Необходимо помнить о следующих ограничениях в использовании CHEK:

- Данные в CHECK берутся только из текущей записи. Не следует брать данные для выражения в CHECK из других записей этой же таблицы – они могут быть изменены другими пользователями.
- Поле может иметь только одно ограничение CHECK.

- Если для описания поля использовался домен, который имеет доменное ограничение CHECK, то его нельзя переопределить на уровне конкретного поля в таблице.

Надо сказать, что CHECK реализованы при помощи системных триггеров, поэтому следует быть осторожным в использовании очень больших условий, которые могут сильно замедлить процессы вставки и обновления записей.

Удаление ограничений

Часто приходится удалять различные ограничения по самым разным причинам. Чтобы удалить ограничение, необходимо воспользоваться предложением ALTER TABLE следующего вида:

```
ALTER TABLE tablename
DROP CONSTRAINT constraintname
```

где constraintname – имя ограничения, которое следует удалять. Если при создании ограничения было задано какое-то имя, то следует им воспользоваться, а если нет, то надо открыть какое-либо средство администрирования InterBase, поискать все связанные с ним ограничения и выяснить, какое системное имя сгенерировал InterBase для искомого ограничения.

Надо отметить, что удалять ограничения может только владелец таблицы или системный администратор SYSDBA.

Представления

Те, кто знаком с языком SQL, не нуждаются в подробных объяснениях этого предмета, но для сохранения порядка изложения приведем все же краткое определение представлений.

> Представление (VIEW) – это виртуальная таблица, созданная на основе запроса к обычным таблицам. Представление реализовано как запрос, хранящийся на сервере и выполняющийся всякий раз, когда происходит обращение к представлению.

Давайте рассмотрим различные варианты использования представлений. Представления дают возможность создать уровни организации данных, позволяющие отделить реализацию хранения данных от их вида. Например, можно создать представление, которое выбирает данные из несколько таблиц. Если клиенты будут использовать это представление, а не напрямую обращаться к лежащим в его основе таблицам, то у разработчика базы данных появляется возможность менять запрос, лежащий в основе представления, изменять его (с целью оптимизации, например), а клиент ничего не будет замечать – для него это будет все то же представление.

Помимо того что они изолируют реализацию хранения данных от пользователя, представления позволяют организовать данные в более удобном и простом виде. Проблема "упрощения" организации данных возникает, когда число таблиц в базе данных становится достаточно большим, а взаимосвязи между ними – сложными. Представление позволяет исключить (или, наоборот, добавить) часть данных, не нужных конкретному клиенту базы данных (или – необходимых).

Также представления позволяют более просто организовать безопасность в базе данных InterBase. Определенные пользователи могут иметь права только на чтение/изменение данных в представлении, но не иметь никаких прав (и даже никакого понятия) о таблицах, лежащих в основе представления! Подробнее о вопросах безопасности в InterBase см. главу "Безопасность в InterBase: пользователи, роли и права" (ч. 4).

Синтаксис DDL для работы с представлениями

Для создания и удаления представлений существуют команды, определенные DDL (Data Definition Language – подмножество SQL, см. глоссарий), которые мы сейчас рассмотрим.

Чтобы создать представление в InterBase, необходимо использовать предложение следующего синтаксиса:

```
CREATE VIEW viewname [(view_column [, view_column…])]
AS <SELECT> [WITH CHECK OPTION];
```

Здесь viewname – имя представления, которое должно быть уникальным в пределах базы данных, далее идет группа не всегда обязательных наименований полей, входящих в представление: [(view_column [, view_column…])]. Обязательно необходимо определить предложение <SELECT>, которое выбирает данные, включаемые в представление. Необязательный параметр WITH CHECK OPTION мы обсудим ниже – в разделе "Модифицируемые представления".

Чтобы изменить какое-либо представление, придется его пересоздать, т. е. удалить и создать заново. При удалении представления необходимо также удалить все зависимые от него объекты – триггеры, хранимые процедуры и другие представления. В этом заключается одно из главных неудобств работы с представлениями – необходимость пересоздавать дерево использующих представление объектов (существуют утилиты, которые позволяют сделать это более "безболезненно", например IBAlterView, см. приложение "Инструменты администратора и разработчика InterBase"). Чтобы удалить представление, необходимо воспользоваться следующей командой DDL:

```
DROP VIEW viewname;
```

Примеры представлений

Вот пример простого представления:

```
CREATE VIEW MyView AS
SELECT NAME, PRICE_1
FROM Table_example;
```

В этом примере мы создаем представление на основе запроса к таблице Table_example, которую мы рассматривали в главе "Таблицы. Первичные ключи и генераторы". В данном случае представление будет состоять из двух полей – NAME и PRICE_1, которые будут выбираться из таблицы Table_example без всяких условий, т. е. число записей в представлении MyView будет равно числу записей в Table_example.

Однако представления не всегда являются такими простыми. Они могут основываться на данных из нескольких таблиц и даже на основе других представлений. Также представления могут содержать данные, получаемые на основе различных выражений – в том числе на основе агрегатных функций.

Чтобы подробнее рассмотреть использование этого применения представлений, давайте создадим две таблицы, связанные отношением один-ко-многим (часто такое отношение называют мастер-деталью или master-detail). Вот DDL-скрипт для создания этих таблиц:

```
/* Table: WISEMEN */
CREATE TABLE WISEMEN (
    ID_WISEMAN INTEGER NOT NULL,
    WISEMAN_NAME VARCHAR(80));
/* Primary keys definition */
ALTER TABLE WISEMEN ADD CONSTRAINT PK_WISEMEN PRIMARY KEY
(ID_WISEMAN);
/* Table: WISEBOOK */
CREATE TABLE WISEBOOK (
    ID_BOOK INTEGER NOT NULL,
    ID_WISEMAN INTEGER,
    BOOK VARCHAR(80));
/* Primary keys definition */
ALTER TABLE WISEBOOK ADD CONSTRAINT PK_WISEBOOK PRIMARY KEY
(ID_BOOK);
/* Foreign keys definition */
```

```
ALTER TABLE WISEBOOK ADD CONSTRAINT FK_WISEBOOK FOREIGN KEY
(ID_WISEMAN) REFERENCES WISEMEN (ID_WISEMAN);
```

Итак, мы создали две таблицы – WISEMEN и WISEBOOK, которые связали между собой отношением master-detail с помощью ограничения внешнего ключа – FOREIGN KEY. Предположим, что эти таблицы будут хранить информацию о великих китайских мудрецах и их произведениях. Теперь мы можем создать несколько представлений на основе этих таблиц. Например, создадим представление, которое показывает, сколько произведений есть у каждого мудреца:

```
CREATE VIEW WiseBookCount
(WISEMAN,
HOW_WISEBOOKS) AS
SELECT M.WISEMAN_NAME, COUNT(B.BOOK)
FROM WISEMEN M, WISEBOOK B
WHERE (M.ID_WISEMAN = B.ID_WISEMAN)
GROUP BY M.WISEMAN_NAME
```

Обратите внимание, что при использовании любых вычисляемых выражений вроде агрегатных функций COUNT(), SUM(), MAX() и т. д., необходимо использовать явное именование полей представления, т. е. давать имена всем полям, возвращаемым запросом. Как видно из этого примера, эти имена не обязательно должны совпадать с именами полей запроса, но их количество должно совпадать с количеством полей, возвращаемых запросом. Установление того, какое поле, возвращаемое запросом, соответствует какому полю представления, осуществляется по порядковому номеру – первое поле запроса отобразится в первое поле представления, второе – во второе и т. д.

А если мы захотим узнать, какой же из мудрецов написал больше всего книг? И попытаемся добавить в запрос, лежащий в основе представления, выражение для сортировки – ORDER BY. Однако эта попытка будет неудачной: использование сортировки ORDER BY в представлениях не допускается и при попытке создать представление с запросом, содержащим ORDER BY, возникнет ошибка. Если мы желаем отсортировать результаты, возвращаемые представлением, то придется это сделать на стороне клиента:

```
SELECT * FROM WiseBookCount ORDER BY HOW_WISEBOOKS
```

Выполнение этого SQL-запроса приведет к желаемому результату.

Помимо ограничения на использование выражения ORDER BY в представлениях, также нельзя использовать в качестве источника данных набор данных, получаемых в результате выполнения хранимых процедур (см. чуть ниже главу "Хранимые процедуры").

Пожалуй, стоит привести еще один пример, иллюстрирующий применение представлений. Предположим, нам необходимо вывести список мудрецов, чье имя начинается с буквы "К". В этом случае нам поможет представление с условиями:

```
CREATE VIEW WiseMen2
(WISEMAN) AS
SELECT M.WISEMAN_NAME
```

```
FROM WISEMEN M
WHERE M.WISEMAN_NAME LIKE 'K%'
```

Таким образом, легко создавать представления, которые исполняют роль постоянно обновляемых поставщиков данных, отбирая их из базы данных по определенным условиям.

Модифицируемые представления

Выше мы упомянули о том, чт. е. возможность создавать изменяемые представления данных. Это действительно так – существует возможность не только читать данные из представления, но и изменять их!

Есть два способа сделать представление модифицируемым. Первый способ применим, когда представление создается на основе единственной таблицы (или другого модифицируемого представления), причем все столбцы данной таблицы должны позволять наличие NULL. При этом запрос, на котором основано представление, не может содержать подзапросов, агрегатных функций, UDF, хранимых процедур, предложений DISTINCT и HAVING. Если выполняются все эти условия, то представление автоматически становится модифицируемым, т. е. для него можно выполнять запросы DELETE, INSERT и UPDATE, которые будут изменять данные в таблице-источнике.

Список условий довольно внушительный и сильно ограничивает применение таких модифицируемых представлений, поэтому они используются относительно редко.

Чтобы сделать модифицируемым представление, которое нарушает любое из вышеперечисленных условий, применяется механизм триггеров. Подробнее о том, что такое триггер, рассказывается в главе "Триггеры" (ч. 1), а сейчас мы лишь рассмотрим общие принципы организации изменения данных во VIEW.

Для реализации обновляемого представления с помощью триггеров необходимо сделать следующее.

Создать 3 триггера для данного представления на события: BEFORE DELETE, BEFORE UPDATE и BEFORE INSERT. В этих триггерах описать, что должно происходить с данными при удалении, изменении и вставке.

Затем следует использовать данное представление в запросах на модификацию – DELETE, INSERT или UPDATE. Когда InterBase примет этот запрос, то проверит, существуют ли для данного представления соответствующие триггеры, т. е. BEFORE DELETE/INSERT/UPDATE. Если триггер для выполняемого действия существует, то InterBase вызовет его для модификации реальных данных в таблицах, лежащих в основе представления (хотя это могут быть и другие данные – каких-либо ограничений на текст этих триггеров нет), а затем перечитает строку (или строки), над которой производилась модификация.

Таким образом, есть возможность реализовать сложные цепочки обновлений данных в представлениях.

В описании синтаксиса создания представления упоминалась опция WITH CHECK OPTION. Если при создании модифицируемого представления будет указана эта опция, то каждая строка данных, вставляемая или изменяемая в этом представлении, будет проверена на условие "попадания" в представление. Это

можно объяснить так: если новая запись, вставляемая пользователем или получившаяся в результате обновления существующей записи, не удовлетворяет условиям запроса, который является поставщиком данных для VIEW, то вставка этой записи будет отменена и возникнет ошибка.

Заключение

Несмотря на кажущуюся простоту создания и использования представлений, они дают большие возможности для улучшения организации данных в базе данных и позволяют создавать иерархию организации данных.

Одни разработчики приложений базы данных очень активно используют представления в своей работе, другие избегают их применения, мотивируя это сложностью модификации представлений и стремлением сохранить схему своей базы данных максимально простой и эффективной. Как вы будете применять представления в своей работе – решать вам. Главное – помнить о существовании такого мощного инструмента, как представление, и уметь им пользоваться.

Хранимые процедуры

Предметом этой главы является один из наиболее мощных инструментов, предлагаемых разработчикам приложений баз данных InterBase для реализации бизнес-логики. Хранимые процедуры (англ. stored procedures) позволяют реализовать значительную часть логики приложения на уровне базы данных и таким образом повысить производительность всего приложения, централизовать обработку данных и уменьшить количество кода, необходимого для выполнения поставленных задач. Практически любое достаточно сложное приложение баз данных не обходится без использования хранимых процедур.

Помимо этих широко известных преимуществ использования хранимых процедур, общих для большинства реляционных СУБД, хранимые процедуры InterBase могут играть роль практически полноценных наборов данных, что позволяет использовать возвращаемые ими результаты в обычных SQL-запросах.

Часто начинающие разработчики представляют себе хранимые процедуры просто как набор специфических SQL-запросов, которые что-то делают внутри базы данных, причем бытует мнение, что работать с хранимыми процедурами намного сложнее, чем реализовать ту же функциональность в клиентском приложении, на языке высокого уровня.

Так что же такое хранимые процедуры в InterBase?

Хранимая процедура (ХП) – это часть метаданных базы данных, представляющая собой откомпилированную во внутреннее представление InterBase подпрограмму, написанную на специальном языке, компилятор которого встроен в ядро сервера InterBase.

Хранимую процедуру можно вызывать из клиентских приложений, из триггеров и других хранимых процедур. Хранимая процедура выполняется внутри серверного процесса и может манипулировать данными в базе данных, а также возвращать вызвавшему ее клиенту (т. е. триггеру, ХП, приложению) результаты своего выполнения.

Основой мощных возможностей, заложенных в ХП, является процедурный язык программирования, имеющий в своем составе как модифицированные предложения обычного SQL, такие, как INSERT, UPDATE и SELECT, так и средства организации ветвлений и циклов (IF, WHILE), а также средства обработки ошибок и исключительных ситуаций. Язык хранимых процедур позволяет реализовать сложные алгоритмы работы с данными, а благодаря ориентированности на работу с реляционными данными ХП получаются значительно компактнее аналогичных процедур на традиционных языках.

Надо отметить, что и для триггеров используется этот же язык программирования, за исключением ряда особенностей и ограничений. Отличия подмножества языка, используемого в триггерах, от языка ХП подробно рассмотрены в главе "Триггеры" (ч. 1).

Пример простой хранимой процедуры

Настало время создать первую хранимую процедуру и на ее примере изучить процесс создания хранимых процедур. Но для начала следует сказать несколько слов о том, как работать с хранимыми процедурами. Дело в том, что своей сла-

вой малопонятного и неудобного инструмента ХП обязаны чрезвычайно бедным стандартным средствам разработки и отладки хранимых процедур. В документации по InterBase рекомендуется создавать процедуры с помощью файлов SQL-скриптов, содержащих текст ХП, которые подаются на вход интерпретатору isql, и таким образом производить создание и модификацию ХП. Если в этом SQL-скрипте на этапе компиляции текста процедуры в BLR (о BLR см. главу "Структура базы данных InterBase" (ч. 4)) возникнет ошибка, то isql выведет сообщение о том, на какой строке файла SQL-скрипта возникла эта ошибка. Исправляйте ошибку и повторяйте все сначала. Про отладку в современном понимании этого слова, т. е. о трассировке выполнения, с возможностью посмотреть промежуточные значения переменных, речь вообще не идет. Очевидно, что такой подход не способствует росту привлекательности хранимых процедур в глазах разработчика.

Однако помимо стандартного минималистского подхода к разработке ХП существуют также инструменты сторонних разработчиков, которые делают работу с хранимыми процедурами весьма удобной. Большинство универсальных продуктов для работы с InterBase, перечисленных в приложении "Инструменты администратора и разработчика InterBase", предоставляют удобный инструментарий для работы с ХП. Мы рекомендуем обязательно воспользоваться одним из этих инструментов для работы с хранимыми процедурами и изложение материала будем вести в предположении, что у вас имеется удобный GUI-инструмент, избавляющий от написания традиционных SQL-скриптов.

Синтаксис хранимых процедур описывается следующим образом:

```
CREATE PROCEDURE name
[( param datatype [, param datatype …])]
[RETURNS ( param datatype [, param datatype …])]
AS
<procedure_body>;
< procedure_body> = [<variable_declaration_list>]
< block>
< variable_declaration_list> =
DECLARE VARIABLE var datatype;
[DECLARE VARIABLE var datatype; …]
<block> =
BEGIN
< compound_statement>
[< compound_statement> …]
END
< compound_statement> = {<block> | statement;}
```

Выглядит довольно объемно и может быть даже громоздко, но на самом деле все очень просто. Для того чтобы постепенно освоить синтаксис, давайте будем рассматривать постепенно усложняющиеся примеры.

Итак, вот пример очень простой хранимой процедуры, которая принимает на входе два числа, складывает их и возвращает полученный результат:

```
CREATE PROCEDURE SP_Add(first_arg DOUBLE PRECISION,
second_arg DOUBLE PRECISION)
RETURNS (Result DOUBLE PRECISION)
AS
```

```
BEGIN
Result=first_arg+second_arg;
SUSPEND;
END
```

Как видите, все просто: после команды CREATE PROCEDURE указывается имя вновь создаваемой процедуры (которое должно быть уникальным в пределах базы данных) – в данном случае SP_Add, затем в скобках через запятую перечисляются входные параметры ХП – first_arg и second_arg – с указанием их типов.

Список входных параметров является необязательной частью оператора CREATE PROCEDURE – бывают случаи, когда все данные для своей работы процедура получает посредством запросов к таблицам внутри тела процедуры.

> В хранимых процедурах используются любые скалярные типы данных InterBase. Не предусмотрено применение массивов и типов, определяемых пользователем, – доменов.

Далее идет ключевое слово RETURNS, после которого в скобках перечисляются возвращаемые параметры с указанием их типов – в данном случае только один – Result.

Если процедура не должна возвращать параметры, то слово RETURNS и список возвращаемых параметров отсутствуют.

После RETURNS() указано ключевое слово AS. До ключевого слова AS идет *заголовок*, а после него – *тело* процедуры.

Тело хранимой процедуры представляет собой перечень описаний ее внутренних (локальных) переменных (если они есть, подробнее рассмотрим ниже), разделяемый точкой с запятой (;), и блок операторов, заключенный в операторные скобки BEGIN END. В данном случае тело ХП очень простое – мы просто складываем два входных аргумента и присваиваем их результат выходному, а затем вызываем команду SUSPEND. Чуть позже мы разъясним суть действия этой команды, а пока лишь отметим, что она нужна для передачи возвращаемых параметров туда, откуда была вызвана хранимая процедура.

Разделители в хранимых процедурах

Обратите внимание, что оператор внутри процедуры заканчивается точкой с запятой (;). Как известно, точка с запятой является стандартным разделителем команд в SQL – она является сигналом интерпретатору SQL, что текст команды введен полностью и надо начинать его обрабатывать. Не получится ли так, что, обнаружив точку с запятой в середине ХП, интерпретатор SQL сочтет, что команда введена полностью и попытается выполнить часть хранимой процедуры? Это предположение не лишено смысла. Действительно, если создать файл, в который записать вышеприведенный пример, добавить команду соединения с базы данных и попытаться выполнить этот SQL-скрипт с помощью интерпретатора isql, то будет возвращена ошибка, связанная с неожиданным, по мнению интерпретатора, окончанием команды создания хранимой процедуры. Если создавать хранимые процедуры с помощью файлов SQL-скриптов, без использования специализированных инструментов разработчика InterBase, то необходимо перед каждой командой создания ХП (то же относится и к триггерам) менять раздели-

тель команд скрипта на другой символ, отличный от точки с запятой, а после текста ХП восстанавливать его обратно. Команда isql, изменяющая разделитель предложений SQL, выглядит так:

```
SET TERM <new_term><old_term>
```

Для типичного случая создания хранимой процедуры это выглядит так:

```
SET TERM ^;
CREATE PROCEDURE some_procedure
…..
END
^
SET TERM ;^
```

Вызов хранимой процедуры

Но вернемся к нашей хранимой процедуре. Теперь, когда она создана, ее надо как-то вызвать, передать ей параметры и получить возвращаемые результаты. Это сделать очень просто – достаточно написать SQL-запрос следующего вида:

```
SELECT *
FROM Sp_add(181.35, 23.09)
```

Этот запрос вернет нам одну строку, содержащую всего одно поле Result, в котором будет находиться сумма чисел 181.35 и 23.09 т. е. 204.44.

Таким образом, нашу процедуру можно использовать в обычных SQL-запросах, выполняющихся как в клиентских программах, так и в других ХП или триггерах. Такое использование нашей процедуры стало возможным из-за применения команды SUSPEND в конце хранимой процедуры.

Дело в том, что в InterBase (и во всех его клонах) существуют два типа хранимых процедур: процедуры-выборки (selectable procedures) и исполняемые процедуры (executable procedures). Отличие в работе этих двух видов ХП заключается в том, что процедуры-выборки обычно возвращают множество наборов выходных параметров, сгруппированных построчно, которые имеют вид набора данных, а исполняемые процедуры могут либо вообще не возвращать параметры, либо возвращать только один набор выходных параметров, перечисленных в Returns, т. е. одну строку параметров. Процедуры-выборки вызываются в запросах SELECT, а исполняемые процедуры – с помощью команды EXECUTE PROCEDURE.

Оба вида хранимых процедур имеют одинаковый синтаксис создания и формально ничем не отличаются, поэтому любая исполнимая процедура может быть вызвана в SELECT-запросе и любая процедура-выборка – с помощью EXECUTE PROCEDURE. Вопрос в том, как поведут себя ХП при разных типах вызова. Другими словами, разница заключается в проектировании процедуры для определенного типа вызова. То есть процедура-выборка специально создается для вызова из запроса SELECT, а исполняемая процедура – для вызова с использованием EXECUTE PROCEDURE. Давайте рассмотрим, в чем же заключаются отличия при проектировании этих двух видов ХП.

Для того чтобы понять, как работает процедура-выборка, придется немного углубиться в теорию. Давайте представим себе обычный SQL-запрос вида

SELECT ID, NAME FROM Table_example. В результате его выполнения мы получаем на выходе таблицу, состоящую из двух столбцов (ID и NAME) и некоторого количества строк (равного количеству строк в таблице Table_example). Возвращаемая в результате этого запроса таблица называется также набором данных SQL. Задумаемся же, как формируется набор данных во время выполнения этого запроса. Сервер, получив запрос, определяет, к каким таблицам он относится, затем выясняет, какое подмножество записей из этих таблиц необходимо включить в результат запроса. Далее сервер считывает каждую запись, удовлетворяющую результатам запроса, выбирает из нее нужные поля (в нашем случае это ID и NAME) и отсылает их клиенту. Затем процесс повторяется снова – и так для каждой отобранной записи.

Все это отступление нужно для того, чтобы уважаемый читатель понял, что все наборы данных SQL формируются построчно, в том числе и в хранимых процедурах! И основное отличие процедур-выборок от исполняемых процедур в том, что первые спроектированы для возвращения множества строк, а вторые – только для одной. Поэтому они и применяются по-разному: процедура-выборка вызывается при помощи команды SELECT, которая "требует" от процедуры отдать все записи, которая она может вернуть. Исполняемая процедура вызывается с помощью EXECUTE PROCEDURE, которая "вынимает" из ХП только одну строку, а остальные (даже если они есть!) игнорирует.

Давайте рассмотрим пример процедуры-выборки, чтобы было понятнее. Для упрощения создадим хранимую процедуру, которая работает точно так же, как запрос SELECT ID, NAME FROM Table_Example, т. е. она просто делает выборку полей ID и NAME из всей таблицы. Вот этот пример:

```
CREATE PROCEDURE Simple_Select_SP
RETURNS (
    procID INTEGER,
    procNAME VARCHAR(80))
AS
BEGIN
  FOR
    SELECT ID, NAME FROM table_example
    INTO :procID, :procNAME
  DO
  BEGIN
    SUSPEND;
  END
END
```

Давайте разберем действия этой процедуры, названной Simple_Select_SP. Как видите, она не имеет входных параметров и имеет два выходных параметра – ID и NAME. Самое интересное, конечно, заключено в теле процедуры. Здесь использована конструкция FOR SELECT:

```
FOR
    SELECT ID, NAME FROM table_example
    INTO :procID, :procNAME
  DO
BEGIN
  /*что-то делаем с переменными procID и procName*/
END
```

Этот кусочек кода означает следующее: для каждой строки, выбранной из таблицы Table_example, поместить выбранные значения в переменные procID и procName, а затем произвести какие-то действия с этими переменными.

Вы можете сделать удивленное лицо и спросить: "Переменные? Какие еще переменные?" Это нечто вроде сюрприза этой главы – то, что в хранимых процедурах мы можем использовать переменные. В языке ХП можно объявлять как собственные локальные переменные внутри процедуры, так и использовать входные и выходные параметры в качестве переменных.

Для того чтобы объявить локальную переменную в хранимой процедуре, необходимо поместить ее описание после ключевого слова AS и до первого слова BEGIN. Описание локальной переменной выглядит так:

```
DECLARE VARIABLE <variable_name> <variable_type>;
```

Например, чтобы объявить целочисленную локальную переменную MyInt, нужно вставить между AS и BEGIN следующее описание:

```
DECLARE VARIABLE MyInt INTEGER;
```

Переменные в нашем примере начинаются с двоеточия. Это сделано потому, что обращение к ним идет внутри SQL-команды FOR SELECT, поэтому для различения полей в таблицах, которые используются в SELECT, и переменных необходимо предварять последние двоеточием. Ведь переменные могут иметь точно такое же название, как и поля в таблицах!

Но двоеточие перед именем переменной необходимо использовать только внутри SQL-запросов. Вне текстов обращение к переменной делается без двоеточия, например:

```
procName='Some name';
```

Но вернемся к телу нашей процедуры. Предложение FOR SELECT возвращает данные не в виде таблицы – набора данных, а по одной строчке. Каждое возвращаемое поле должно быть помещено в свою переменную: ID => procID, NAME => procName. В части DO эти переменные посылаются клиенту, вызвавшему процедуру, с помощью команды SUSPEND.

Таким образом, команда FOR SELECT… DO организует цикл по записям, выбираемым в части SELECT этой команды. В теле цикла, образуемого частью DO, выполняется передача очередной сформированной записи клиенту с помощью команды SUSPEND.

Итак, процедура-выборка предназначена для возвращения одной или более строк, для чего внутри тела ХП организуется цикл, заполняющий результирующие параметры-переменные. И в конце тела этого цикла обязательно стоит команда SUSPEND, которая вернет очередную строку данных клиенту.

Циклы и операторы ветвления

Помимо команды FOR SELECT... DO, организующей цикл по записям какой-либо выборки, существует другой вид цикла – WHILE...DO, который позволяет организовать цикл на основе проверки любых условий. Вот пример ХП, использующей цикл WHILE…DO. Эта процедура возвращает квадраты целых чисел от 0 до 99:

```
CREATE PROCEDURE QUAD
RETURNS (QUADRAT INTEGER)
AS
DECLARE VARIABLE I INTEGER;
BEGIN
  I=1;
  WHILE (i<100) DO
   BEGIN
      QUADRAT= I*I;
       I=I+1;
      SUSPEND;
   END
END
```

В результате выполнения запроса SELECT * FROM QUAD мы получим таблицу, содержащую один столбец QUADRAT, в котором будут квадраты целых чисел от 1 до 99.

Помимо перебора результатов SQL-выборки и классического цикла, в языке хранимых процедур используется оператор IF...THEN..ELSE, позволяющий организовать ветвление в зависимости от выполнения каких-либо условий. Его синтаксис похож на большинство операторов ветвления в языках программирования высокого уровня, вроде Паскаля и Си.

Давайте рассмотрим более сложный пример хранимой процедуры, которая делает следующее:

1. Вычисляет среднюю цену в таблице Table_example (см. глава "Таблицы. Первичные ключи и генераторы")
2. Далее для каждой записи в таблице делает следующую проверку: если существующая цена (PRICE) больше средней цены, то устанавливает цену, равную величине средней цены, плюс задаваемый фиксированный процент.
3. Если существующая цена меньше или равна средней цене, то устанавливает цену, равную прежней цене, плюс половина разницы между прежней и средней ценой.
4. Возвращает все измененные строки в таблице.

Для начала определим имя ХП, а также входные и выходные параметры. Все это прописывается в заголовке хранимой процедуры:

```
CREATE PROCEDURE IncreasePrices (
Percent2Increase DOUBLE PRECISION)
RETURNS (ID INTEGER, NAME VARCHAR(80), new_price DOUBLE
PRECISION) AS
```

Процедура будет называться IncreasePrices, у нее один входной параметр Percent2Increase, имеющий тип DOUBLE PRECISION, и 3 выходных параметра – ID, NAME и new_price. Обратите внимание, что первые два выходных параметра имеют такие же имена, как и поля в таблице Table_example, с которой мы собираемся работать. Это допускается правилами языка хранимых процедур.

Теперь мы должны объявить локальную переменную, которая будет использоваться для хранения среднего значения. Это объявление будет выглядеть следующим образом:

```
DECLARE VARIABLE avg_price DOUBLE PRECISION;
```

Теперь перейдем к телу хранимой процедуры. Откроем тело ХП ключевым словом BEGIN.

Сначала нам необходимо выполнить первый шаг нашего алгоритма – вычислить среднюю цену. Для этого мы воспользуемся запросом следующего вида:

```
SELECT AVG(Price_1)
FROM Table_Example
INTO :avg_price;
```

Этот запрос использует агрегатную функцию AVG, которая возвращает среднее значение поля PRICE_1 среди отобранных строк запроса – в нашем случае среднее значение PRICE_1 по всей таблице Table_example. Возвращаемое запросом значение помещается в переменную avg_price. Обратите внимание, что переменная avg_price предваряется двоеточием – для того, чтобы отличить ее от полей, используемых в запросе.

Особенностью данного запроса является то, что он всегда возвращает строго одну-единственную запись. Такие запросы называются singleton-запросами. И только такие выборки можно использовать в хранимых процедурах. Если запрос возвращает более одной строки, то его необходимо оформить в виде конструкции FOR SELECT...DO, которая организует цикл для обработки каждой возвращаемой строки.

Итак, мы получили среднее значение цены. Теперь необходимо пройтись по всей таблице, сравнить значение цены в каждой записи со средней ценой и предпринять соответствующие действия.

Сначала организуем перебор каждой записи из таблицы Table_example:

```
FOR
        SELECT ID, NAME, PRICE_1
        FROM Table_Example
        INTO :ID, :NAME, :new_price
DO
       BEGIN
      /*здесь обрабатываем каждую запись*/
       END
```

При выполнении этой конструкции из таблицы Table_example построчно будут выниматься данные и значения полей в каждой строке будут присвоены переменным ID, NAME и new_price. Вы, конечно, помните, что эти переменные объявлены как выходные параметры, но беспокоиться, что выбранные данные будут возвращены как результаты, не стоит: тот факт, что выходным параметрам что-либо присвоено, не означает, что вызывающий ХП клиент немедленно получит эти значения! Передача параметров осуществляется только при исполнении команды SUSPEND, а до этого мы можем использовать выходные параметры в качестве обычных переменных – в нашем примере мы именно так и делаем с параметром new_price.

Итак, внутри тела цикла BEGIN…END мы можем обработать значения каждой строки. Как вы помните, нам необходимо выяснить, как существующая цена соотносится со средней, и предпринять соответствующие действия. Эту процедуру сравнения мы реализуем с помощью оператора IF:

```
IF (new_price > avg_price) THEN /*если существующая цена больше
средней цены*/
 BEGIN
/*то установим новую цену, равную величине средней цены, плюс
фиксированный процент */
  new_price = (avg_price + avg_price*(Percent2Increase/100));
  UPDATE Table_example
   SET PRICE_1 = :new_price
   WHERE ID = :ID;
END
ELSE
 BEGIN
 /* Если существующая цена меньше или равна средней цене, то
установим цену, равную прежней цене, плюс половина разницы меж-
ду прежней и средней ценой */
  new_price = (new_price + ((avg_price - new_price)/2));
  UPDATE Table_example
   SET PRICE_1 = :new_price
   WHERE ID = :ID;
END
```

Как видите, получилось достаточно большая конструкция IF, в которой трудно было бы разобраться, если бы не комментарии, заключенные в символы /**/.

Для того чтобы изменить цену в соответствии с вычисленной разницей, мы воспользуемся оператором UPDATE, который позволяет модифицировать существующие записи – одну или несколько. Для того чтобы однозначно указать, в какой записи нужно изменять цену, мы используем в условии WHERE поле первичного ключа, сравнивая его со значением переменной, в которой хранится значение ID для текущей записи: ID=:ID. Обратите внимание, что переменная ID предваряется двоеточием.

После выполнения конструкции IF...THEN...ELSE в переменных ID, NAME и new_price находятся данные, которые мы должны возвратить клиенту, вызвавшему процедуру. Для этого после IF необходимо вставить команду SUSPEND, которая перешлет данные туда, откуда вызвали ХП. На время пересылки действие процедуры будет приостановлено, а когда от ХП потребуется новая запись, то она будет вновь продолжена, – и так будет продолжаться до тех пор, пока FOR SELECT...DO не переберет все записи своего запроса.

Надо отметить, что помимо команды SUSPEND, которая только приостанавливает действие хранимой процедуры, существует команда EXIT, которая прекращает хранимую процедуру после передачи строки. Однако командой EXIT пользуются достаточно редко, поскольку она нужна в основном для того, чтобы прервать цикл при достижении какого-либо условия.

При этом в случае, когда процедура вызывалась оператором SELECT и завершена по EXIT, последняя извлеченная строка не будет возвращена. То есть,

если вам нужно прервать процедуру и все-таки получить эту строку, надо воспользоваться последовательностью

```
SUSPEND;
EXIT;
```

Основное назначение EXIT – получение singleton-наборов данных, возвращаемых параметров путем вызова через EXECUTE PROCEDURE. В этом случае устанавливаются значения выходных параметров, но из них не формируется набор данных SQL, и выполнение процедуры завершается.

Давайте запишем текст нашей хранимой процедуры полностью, чтобы иметь возможность охватить ее логику одним взглядом:

```
CREATE PROCEDURE IncreasePrices (
Percent2Increase DOUBLE PRECISION)
RETURNS (ID INTEGER, NAME VARCHAR(80),
new_price DOUBLE PRECISION) AS
DECLARE VARIABLE avg_price DOUBLE PRECISION;
BEGIN
SELECT AVG(Price_1)
FROM Table_Example
INTO :avg_price;
FOR
        SELECT ID, NAME, PRICE_1
        FROM Table_Example
        INTO :ID, :NAME, :new_price
DO
       BEGIN
    /*здесь обрабатываем каждую запись*/
IF (new_price > avg_price) THEN /*если существующая цена больше
средней цены*/
 BEGIN
/*установим новую цену, равную величине средней цены, плюс фик-
сированный процент */
  new_price = (avg_price + avg_price*(Percent2Increase/100));
  UPDATE Table_example
   SET PRICE_1 = :new_price
   WHERE ID = :ID;
END
ELSE
 BEGIN
 /* Если существующая цена меньше или равна средней цене, то
устанавливает цену, равную прежней цене, плюс половина разницы
между прежней и средней ценой */
  new_price = (new_price + ((avg_price - new_price)/2));
  UPDATE Table_example
   SET PRICE_1 = :new_price
   WHERE ID = :ID;
END
SUSPEND;
END
END
```

Данный пример хранимой процедуры иллюстрирует применение основных конструкций языка хранимых процедур и триггеров. Далее мы рассмотрим способы применения хранимых процедур для решения некоторых часто возникающих задач.

Рекурсивные хранимые процедуры

Хранимые процедуры InterBase могут быть рекурсивными. Это означает, что из хранимой процедуры можно вызвать саму себя. Допускается до 1000 уровней вложенности хранимых процедур, однако надо помнить о том, что свободные ресурсы на сервере могут закончиться раньше, чем будет достигнута максимальная вложенность ХП.

Одно из распространенных применений хранимых процедур – это обработка древовидных структур, хранящихся в базе данных. Деревья часто используются в задачах состава изделия, складских, кадровых и в других распространенных приложениях.

Давайте рассмотрим пример хранимой процедуры, которая выбирает все товары определенного типа, начиная с определенного уровня вложенности.

Пусть у нас есть следующая постановка задачи: имеем справочник товаров с иерархической структурой такого вида:

```
-Товары
– Бытовая техника
-  Холодильники
                  -  Трехкамерные
                  -  Двухкамерные
                  -  Однокамерные
–  Стиральные машины
                  -  Вертикальные
                  -  Фронтальные
                        -  Классические
                        -  Узкие
–  Компьютерная техника
      …..
```

Эта структура справочника категорий товаров может иметь ветки различной глубины, а также нарастать со временем. Наша задача – обеспечить выборку всех конечных элементов из справочника с "разворачивание полного имени", начиная с любого узла. Например, если мы выбираем узел "Стиральные машины", то нам надо получить следующие категории:

```
Стиральные машины – Вертикальные
Стиральные машины – Фронтальные – Классические
Стиральные машины – Фронтальные – Узкие
```

Определим структуру таблиц для хранения информации справочника товаров. Используем упрощенную схему для организации дерева в одной таблице:

```
CREATE TABLE GoodsTree
(ID_GOOD INTEGER NOT NULL,
 ID_PARENT_GOOD INTEGER,
 GOOD_NAME VARCHAR(80),
 constraint pkGood primary key (ID_GOOD));
```

Создаем одну таблицу GoodsTree, в которой всего 3 поля: ID_GOOD – уникальный идентификатор категории, ID_PARENT_GOOD – идентификатор категории-родителя для данной категории и GOOD_NAME – наименование категории. Чтобы обеспечить целостность данных в этой таблице, наложим на эту таблицу ограничение внешнего ключа:

```
ALTER TABLE GoodsTree
ADD CONSTRAINT FK_goodstree
FOREIGN KEY (ID_PARENT_GOOD)
REFERENCES GOODSTREE(ID_GOOD)
```

Таблица ссылается сама на себя и данный внешний ключ следит за тем, чтобы в таблице не было ссылок на несуществующих родителей, а также препятствует попыткам удалить категории товаров, у которых есть потомки.

Давайте занесем в нашу таблицу следующие данные:

ID_GOOD	ID_PARENT_GOOD	GOOD_NAME
1	0	GOODS
2	1	Бытовая техника
3	1	Компьютеры и комплектующие
4	2	Холодильники
5	2	Стиральные машины
6	4	Трехкамерные
7	4	Двухкамерные
8	4	Однокамерные
9	5	Вертикальные
10	5	Фронтальные
11	10	Узкие
12	10	Классические

Теперь, когда у нас есть место для хранения данных, мы можем приступить к созданию хранимой процедуры, выполняющей вывод всех "окончательных" категорий товаров в "развернутом" виде – например, для категории "Трехкамерные" полное имя категории будет выглядеть как "Бытовая техника Холодильники Трехкамерные".

В хранимых процедурах, обрабатывающих древообразные структуры, сложилась своя терминология. Каждый элемент дерева называются узлом; а отношения между ссылающимися друг на друга узлами называется отношениями родитель-потомок. Узлы, находящиеся на самом конце дерева и не имеющие потомков, называются "листьями".

У нашей хранимой процедуры входным параметром будет идентификатор категории, начиная с которого мы должны будем начать развертку. Хранимая процедура будет иметь следующий вид:

```
CREATE PROCEDURE GETFULLNAME (ID_GOOD2SHOW INTEGER)
RETURNS (FULL_GOODS_NAME VARCHAR(1000),
ID_CHILD_GOOD INTEGER)
AS
```

```
DECLARE VARIABLE CURR_CHILD_NAME VARCHAR(80);
BEGIN
/*Организуем внешний цикл FOR SELECT по непосредственным потом-
кам товара с ID_GOOD=ID_GOOD2SHOW */
 FOR SELECT gt1.id_good, gt1.good_name
     FROM GoodsTree gt1
     WHERE gt1.id_parent_good=:ID_good2show
     INTO:ID_CHILD_GOOD, :full_goods_name
 DO
    BEGIN
   /*Проверка с помощью функции EXISTS, которая возвращает
TRUE, если запрос в скобках вернет хотя бы одну строку. Если у
найденного узла с ID_PARENT_GOOD = ID_CHILD_GOOD нет потомков,
то он является "листом" дерева и  попадает в результаты */
    IF (NOT EXISTS(
               SELECT * FROM GoodsTree
               WHERE GoodsTree.id_parent_good=:id_child_good))
    THEN
    BEGIN
/* Передаем "лист" дерева в результаты */
     SUSPEND;
    END
     ELSE
/* Для узлов, у которых есть потомки*/
     BEGIN
/*сохраняем имя узла-родителя во временной переменной */
       CURR_CHILD_NAME=full_goods_name;
/* рекурсивно запускаем эту процедуру */
       FOR
           SELECT ID_CHILD_GOOD, full_goods_name
           FROM GETFULLNAME (:ID_CHILD_GOOD)
           INTO:ID_CHILD_GOOD, :full_goods_name
       DO BEGIN
 /*добавляем  имя узла-родителя к найденному имени потомка
с помощью операции конкатенации строк || */
         full_goods_name=CURR_CHILD_NAME||' '||full_goods_name;
         SUSPEND; /* возвращаем полное имя товара*/
       END
     END
     END
END
```

Если мы выполним данную процедуру с входным параметром ID_GOOD2SHOW=1, то получим следующее:

FULL_GOODS_NAME	**ID_CHILD_GOOD**
Бытовая техника Холодильники Трехкамерные	6
Бытовая техника Холодильники Двухкамерные	7
Бытовая техника Холодильники Однокамерные	8
Бытовая техника Стиральные машины Вертикальные	9
Бытовая техника Стиральные машины Фронтальные	11

```
FULL_GOODS_NAME                                 ID_CHILD_GOOD
Узкие
Бытовая техника Стиральные машины Фронтальные  12
Классические
Компьютеры и комплектующие                      3
```

Как видите, с помощью рекурсивной хранимой процедуры мы прошлись по всему дереву категорий и вывели полное наименование категорий-"листьев", которые находятся на самых кончиках ветвей.

Заключение

На этом закончим рассмотрение основных возможностей языка хранимых процедур. Очевидно, что полностью освоить разработку хранимых процедур при чтении одной главы невозможно, однако здесь мы постарались представить и объяснить основные концепции, связанные с хранимыми процедурами. Описанные конструкции и приемы проектирования ХП могут быть применены в большинстве приложений баз данных.

Часть важных вопросов, связанных с разработкой хранимых процедур, будет раскрыта в следующей главе – "Расширенные возможности языка хранимых процедур InterBase", которая посвящена обработке исключений, разрешению ошибочных ситуаций в хранимых процедурах и работе с массивами.

Расширенные возможности языка хранимых процедур InterBase

Эта глава посвящена тем возможностям языка хранимых процедур InterBase, которые позволяют эффективно реализовывать бизнес-логику на уровне базы данных и разрабатывать устойчивые и высокопроизводительные приложения баз данных.

Обработка исключений и ошибок

Исключения

Первой из рассматриваемых особенностей языка хранимых процедур (ХП) и триггеров InterBase является возможность использовать "исключения".

Исключения InterBase во многом похожи на исключения других языков высокого уровня, однако имеют свои особенности. Фактически исключение InterBase – это сообщение об ошибке, которое имеет собственное, задаваемое программистом имя и текст сообщения об ошибке. Создается исключение следующим образом:

```
CREATE EXCEPTION <имя_исключения> <текст_исключения>;
```

Например, мы можем создать исключение такого вида:

```
CREATE EXCEPTION test_except 'Test exception';
```

Исключение легко удалить или изменить – удаление совершается командой DROP EXCEPTION <имя_удаляемого_исключения>, а изменение – ALTER EXCEPTION <имя_исключения> <текст_исключения>.

Чтобы использовать исключение в хранимой процедуре или триггере, необходимо воспользоваться командой следующего вида:

```
EXCEPTION <имя_исключения>;
```

Давайте рассмотрим применение исключений на простом примере хранимой процедуры, выполняющей деление одного числа на другое и возвращающей результат. Нам необходимо отследить случай деления на нуль и возбудить исключение, если делитель равен нулю.

Для нашего примера создадим следующее исключение:

```
CREATE EXCEPTION zero_divide 'Cannot divide by zero!';
```

Создадим хранимую процедуру, использующую это исключение:

```
CREATE PROCEDURE SP_DIVIDE (
    DELIMOE DOUBLE PRECISION,
    DELITEL DOUBLE PRECISION)
RETURNS (
    RESULT DOUBLE PRECISION)
AS
BEGIN
  if (Delitel<0.0000001) then
```

```
    BEGIN
     EXCEPTION zero_divide;
     Result=0;
    END
   ELSE
    BEGIN
      Result=Delimoe/Delitel;
    END
  SUSPEND;
END
```

Как видите, текст ХП тривиален – на входе получаем Delitel и Delimoe, затем сравниваем Delitel с 0.0000001, т. е. фактически с нулем, в пределах выбранной погрешности в одну десятимиллионную (так как вещественные числа невозможно непосредственно сравнивать из-за погрешностей в дробной части). Если Delitel близок к нулю в пределах выбранной погрешности, то мы возбуждаем исключение zero_divide. Что же происходит в случае возникновения исключения? Если мы попробуем вызвать исключение, выполняя процедуру SP_divide с нулевым делителем в isql, то получим следующее:

```
SQL> select * from sp_divide(300,0);
                  RESULT
======================
Statement failed, SQLCODE = -836
exception 1
-Cannot divide by zero!
```

Если мы вызовем эту ХП с нулевым делителем в каком-либо другом приложении, то скорее всего получим сообщение об ошибке следующего вида:

Рис. 1.4. Сообщение о возникновении исключения

Другими словами, сообщение об ошибке – это результат обработки нашего исключения сервером InterBase. Когда InterBase обнаруживает возникшее в ХП или триггере исключение, он прерывает работу этой хранимой процедуры и откатывает все изменения, сделанные в текущем блоке BEGIN…END, причем если ХП является процедурой-выборкой, то отменяются действия лишь до последнего оператора SUSPEND.

Это значит, что если в процедуре-выборке есть цикл, в котором производятся какие-то действия, и в теле цикла есть SUSPEND, то при возбуждении исключения отменятся все действия, выполненные в этом цикле до последнего оператора SUSPEND.

Надо сказать, что в исключениях было бы мало пользы, если бы у разработчика СУБД не было возможности обработать их на уровне базы данных. Чтобы разработчик смог обработать возникшее исключение, применяется следующая конструкция языка ХП и триггеров:

```
WHEN EXCEPTION <имя_исключения> DO
BEGIN
 /*обработка исключения*/
END
```

Использование этой конструкции помогает избежать возвращения стандартной ошибки с текстом исключения и произвести собственные действия по обработке исключения.

Когда возбуждается исключение, происходит следующее: выполнение хранимой процедуры (или триггера) прерывается, InterBase начинает искать конструкцию WHEN EXCEPTION…DO для обработки возникшего исключения в текущем блоке BEGIN...END. Если не находит, то поднимается на уровень выше (выше в том смысле, если имеются вложенные блоки BEGIN…END или когда одна ХП вызвана из другой) и ищет обработчик исключения там и т. д., пока либо не найдет подходящий обработчик исключения, либо не закончится вложенность уровней хранимой процедуры. Если обработчик исключения так и не был найден, то возвращается стандартное сообщение об ошибке, включающее текст исключения. Если обработчик найден, то выполняются действия в его блоке BEGIN...END и управление передается на первый оператор, следующий за END обработчика.

Давайте рассмотрим пример обработки исключения, возбуждаемого в нашей процедуре SP_DIVIDE. Предположим, что мы имеем некоторую внешнюю процедуру SP_divide_all, вызывающую SP_DIVIDE для того, чтобы поделить два числа. Конечно, пример сильно утрирован, но он позволяет пояснить способ и смысл использования исключений.

Итак, вот текст нашей хранимой процедуры:

```
CREATE PROCEDURE sp_test_except(Delitel DOUBLE PRECISION)
RETURNS  (rslt DOUBLE PRECISION, status VARCHAR(50))
AS
BEGIN
  Status='Everything is Ok';
  SELECT result FROM sp_divide(12,:Delitel) INTO :rslt;
  SUSPEND;
  WHEN EXCEPTION Zero_divide DO
   BEGIN
       Status='zero value found!';
       rslt=-1;
       SUSPEND;
   END
END
```

Эта процедура вызывает процедуру SP_DIVIDE. Если параметр Delitel не равен нулю, то процедура SP_DIVIDE выполняется без проблем и в возвращаемое значение rslt помещается частное от деления, а статусная переменная status принимает значение 'Everything is Ok'. В случае, если возникла исключительная ситуация деления на нуль, то результирующая переменная rslt будет равна –1, а в переменной status будет содержаться сообщение об ошибке – 'zero value found!'. Разумеется, в обработчике исключения можно произвести и более слож-

ную обработку, например записать некорректные данные в особую таблицу или попытаться изменить данные для выполнения операции и вновь попробовать ее выполнить и т. д.

Обработка ошибок SQL и InterBase

Разобравшись с обработкой исключений, определяемых пользователем, можем перейти к обработке ошибок InterBase. Ошибка – это фактически то же самое исключение, только возбуждаемое InterBase. Принцип обработки ошибок тот же самый, что и у исключений: если возникает какая-то ошибка, то сервер ищет ее обработчик, последовательно просматривая все уровни вложенности (если они есть) хранимой процедуры, начиная с того уровня, на котором возникла ошибка.

Если обработчик найден, то выполняется код внутри его, а затем управление передается на первый оператор за обработчиком исключений.

Конструкция, с помощью которой производится обработка ошибок, такая же, как и для обработки исключений, только вместо EXCEPTION стоит либо GDSCODE, либо SQLCODE:

```
WHEN GDSCODE|SQLCODE <код_ошибки> DO
BEGIN
 /*обрабатываем ошибку*/
END
```

В зависимости от того, стоит ли в конструкции обработки ошибок GDSCODE или SQLCODE, обрабатываются различные ошибки. Если стоит SQLCODE, то обрабатываются ошибки SQL, а если GDSCODE – то ошибки InterBase. Примером ошибки SQL является ошибка с SQLCODE=-802 "Arithmetic exception, numeric overflow, or string truncation" или SQLCODE=-817 "Attempted update during read-only transaction". Список ошибок SQL и соответствующих им значений SQLCODE приведен в таблице "SQLCODE codes and messages" в [2, гл. 6].

Примером ошибки InterBase является ошибка isc_bad_dbkey 335544322L "invalid database key". Список этих ошибок и их кодов приведен там же – [2, гл. 6].

Таким образом, внутри хранимой процедуры можно "перехватить" практически любую ошибку и корректно на нее отреагировать. Можно перечислять обработчики ошибок один за другим, чтобы определить реакцию на разные ошибки.

А что делать, если нужно прореагировать на любую ошибку? Не прописывать же обработчики всех сотен возможных ошибок? Конечно же, нет. Для того чтобы написать безусловный обработчик, реагирующий на любую ошибку – SQL, InterBase или исключение, следует воспользоваться конструкцией WHEN DO с ключевым словом ANY:

```
WHEN ANY DO
BEGIN
 /*действия при любой нестандартной операции – ошибке
или исключении */
END
```

С помощью использования описанных механизмов можно предусмотреть развитые механизмы обработки ошибок, которые сделают приложения баз данных значительно более устойчивыми.

Работа с массивами в хранимых процедурах

Массивы, как было сказано в главе "Типы данных", позволяют хранить в одном поле набор данных какого-нибудь одного элементарного типа. Однако "простым" SQL-запросом данные не извлечь и не изменить. Необходим особый подход для работы с массивами InterBase в клиентских приложениях – см. раздел "Поддержка array-полей в FIBPlus" в главе (ч. 2) и "Разработка клиентских приложений СУБД InterBase с использованием технологии Microsoft OLE DB" (ч. 3). К тому же не все библиотеки доступа поддерживают работу с массивами.

Хранимые процедуры позволяют легко просматривать данные из массивов. Они используют простой и очевидный синтаксис для обращения к элементам массива. Например, пусть у нас будет таблица, содержащая поле типа массив целых чисел:

```
CREATE TABLE table_array(
ID_table INTEGER,
Array1 INTEGER[3,2]);
```

Теперь можно продемонстрировать, как можно просмотреть данные из этого массива. Для этого внутри хранимой процедуры применяется конструкция вида

```
SELECT Array1[:i, :j] FROM table_array INTO :ElemValue;
```

Давайте оформим механизм доступа к массиву в виде следующей хранимой процедуры:

```
CREATE PROCEDURE SelectFromArr(ID_row INTEGER,
  x INTEGER, y INTEGER, vl integer)
Returns (ElemValue INTEGER)
AS
BEGIN
SELECT array1[:x,:y]
 FROM table_array
 WHERE id_table=:ID_row
 INTO :ElemValue;
 SUSPEND;
END
```

Как видите, текст ХП очевиден – просто извлекаем нужные значения из элемента массива с заданными индексами x и y в строке таблицы table_array с идентификатором ID_table=ID_row. К сожалению, такой синтаксис доступа к элементам массива доступен только внутри хранимых процедур и только для чтения. Для заполнения массива придется воспользоваться программой, применяющей либо InterBase API, либо библиотеки доступа, поддерживающие работу с массивами InterBase – FIBPlus для Delphi/C++Builder/Kylix или IBProvider для продуктов Microsoft.

Заключение

В этой главе был рассмотрен ряд дополнительных возможностей языка хранимых процедур (и триггеров) СУБД InterBase. Использование описанных конструкций языка позволит разрабатывать более развитые и надежные хранимые процедуры, а значит, более быстрые и надежные приложения баз данных.

Триггеры

Триггеры – одно из замечательнейших изобретений разработчиков баз данных. Триггеры позволяют придать "активность" данным, хранящимся в базе данных, централизовать их обработку и упростить логику клиентских приложений.

Что же такое триггер?

Триггер в InterBase – это особый вид хранимой процедуры, которая выполняется автоматически при вставке, удалении или модификации записи таблицы или представления (view). Триггеры могут "срабатывать" непосредственно до или сразу же после указанного события.

Может быть, это звучит достаточно сложно, однако, как это бывает во многих случаях, сама идея, лежащая в основе триггеров, очень проста.

Как вы знаете, SQL дает возможность нам вставлять, удалять и модифицировать данные в таблицах базы данных при помощи соответствующих команд – INSERT, DELETE и UPDATE. Согласитесь, что было бы неплохо иметь возможность перехватить передаваемую команду и что-нибудь сделать с данными, которые добавляются, удаляются или изменяются. Например, записать эти данные в специальную табличку, а заодно записать, кто и когда произвел операцию над данной таблицей. Или сразу же проверить вставляемые данные на какое-нибудь хитрое условие, которое невозможно реализовать с помощью опции CHECK (см. выше главу "Ограничения базы данных"), и в зависимости от результатов проверки принять проводимые изменения или отвергнуть их; изменить эти данные на основании какого-либо запроса или изменить данные в других связанных таблицах.

Вот для того, чтобы выполнять какие-либо действия, связанные с изменением данных в базе данных, и существуют триггеры.

Фактически триггер представляет собой набор команд процедурного языка InterBase, который исполняется при выполнении операций INSERT/DELETE/UPDATE. В отличие от хранимых процедур, триггер никогда ничего не возвращает (да и некому возвращать, ведь триггер явно не вызывается). По той же причине он не имеет также входных параметров, но вместо них имеет контекстные переменные NEW и OLD. Эти переменные позволяют получить доступ к полям таблицы, к которой присоединен триггер (мы расскажем об этом чуть позже).

Триггеру предназначена роль виртуального цензора, который просматривает "письма" и который волен сделать все, что угодно, – пропустить их неизменными, подправить их, просигнализировать об ошибках или даже "доложить об этом" кому следует.

Триггер всегда привязан к какой-то определенной таблице или представлению и может "перехватывать" данные только этой таблицы. Давайте рассмотрим классификацию триггеров и назначение каждого вида. Как уже было сказано, существует 3 основных SQL-операции, применимые к данным, – INSERT/DELETE/UPDATE. Соответственно первое разделение триггеров – по обслуживаемым операциям. Каждый конкретный триггер привязан к какой-либо операции, т. е. триггер срабатывает, когда в "его" таблице происходит данная операция.

> В клоне Yaffil 1.0 реализована поддержка универсальных триггеров, срабатывающих при любой операции.

Также срабатывание триггера может происходить "до" и "после" операции. Таким образом, мы получаем 6 возможных видов триггеров на таблицу – до и после каждой из трех возможных SQL-операций.

Пример триггера

Давайте рассмотрим простой пример триггера, который срабатывает ДО ВСТАВКИ в таблицу и заполняет поле первичного ключа. Мы воспользуемся в качестве основы для триггера таблицей из примера в главе "Таблицы. Первичные ключи и генераторы" этой части:

```
CREATE TABLE Table_example (
    ID INTEGER NOT NULL,
    NAME VARCHAR(80),
    PRICE_1 DOUBLE PRECISION,
    CONSTRAINT pkTable PRIMARY KEY (ID));
```

Здесь поле ID является первичным ключом и значения этого поля должны быть уникальными в пределах таблицы. Чтобы обеспечить выполнение этого требования, создадим генератор и триггер, который будет получать значение генератора и подставлять его в таблицу. Таким образом, в поле ID всегда будут уникальные значения, так как значение генератора будет увеличиваться каждый раз при обращении к триггеру. Итак, создаем генератор:

```
CREATE GENERATOR GEN_TABLE_EXAMPLE_ID;
```

И устанавливаем его начальное значение в единицу:

```
SET GENERATOR GEN_TABLE_EXAMPLE_ID TO 1;
```

Теперь необходимо создать триггер. Надо сказать, что триггер, как и хранимая процедура, может содержать в своем теле несколько операторов, разделенных точкой с запятой. Поэтому если вы не используете один из инструментов, рекомендованных в приложении "Инструменты администратора и разработчика InterBase", а работаете с isql, то вам необходимо воспользоваться командой смены разделителя команд SET TERM, как это было описано в главе "Хранимые процедуры". Мы же будем приводить тексты триггеров без обрамления командами смены разделителя.

Итак, рассмотрим текст нашего триггера:

```
CREATE TRIGGER Table_example_bi FOR Table_example
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.ID IS NULL) THEN
    NEW.ID = GEN_ID(GEN_TABLE_EXAMPLE_ID,1);
END
```

Как видите, триггер очень напоминает хранимую процедуру (фактически, как уже было сказано, это и есть особая разновидность ХП), но есть и несколько отличий. Давайте подробно разберем "строение" триггера.

Описание команды создания триггера начинается с ключевых слов CREATE TRIGGER, после которых следует имя триггера – Table_example_bi. Потом сле-

дует ключевое слово FOR, после которого указано имя таблицы, для которой создается триггер, – Table_example.

На второй строке команды приводится описание сущности триггера – ключевое слово ACTIVE указывает, что триггер является "активным". Триггер также может быть переведен в состояние INACTIVE. Это означает, что он будет храниться в базе данных, но он не будет срабатывать. Сочетание ключевых слов BEFORE INSERT определяет, что триггер срабатывает ДО ВСТАВКИ; а ключевое слово POSITION и число 0 указывают очередность (позицию) создаваемого триггера среди триггеров того же типа для данной таблицы. Позиция триггера нужна потому, что в InterBase возможно создать более 32000 триггеров каждого вида (например, BEFORE INSERT или AFTER UPDATE), и серверу нужно указать, в каком порядке эти триггеры будут выполняться. Триггеры с меньшей позицией выполняются первыми. Если имеется несколько триггеров с одинаковой позицией, то они будут выполняться в алфавитном порядке.

Все рассмотренное выше до ключевого слова AS образует заголовок триггера. После AS следует тело триггера. Собственно в теле и осуществляется вставка значения в поле первичного ключа. Но сначала с помощью уже знакомого вам оператора IF…THEN проверяется, не было ли заполнено это поле на клиенте. Выражение проверки, возвращающей булеву TRUE (истина) или FALSE (ложь), выглядит так:

```
NEW.ID IS NULL
```

"Интересно, что такое NEW?" – спросите вы. Это одна из особенностей, присущая только триггерам, – *контекстная переменная*. Давайте взглянем, как она действует.

Контекстные переменные

Как уже говорилось выше, триггер похож на цензора, бесцеремонно досматривающего все, что относится к интересующему его предмету. Интерес нашего триггера-"цензора" описан сочетанием ключевых слов BEFORE INSERT – это значит, что все операции вставки (INSERT) вызовут срабатывание триггера. Причем он сработает ДО (BEFORE) того, как вставка физически осуществлена. То есть в момент срабатывания триггера данные, присланные кем-либо на вставку, еще не занесены в таблицу. Они находятся в некотором промежуточном буфере. И у триггера есть возможность обращаться к этому буферу, чтобы проверить и/или изменить значения данных-кандидатов на вставку. Эта возможность реализована с помощью контекстной переменной NEW. Можно рассматривать эту переменную как структуру (что-то подобное struct в Си или record в Pascal), элементы которой представляют собой значения, присланные для осуществления операции (INSERT в нашем примере). То есть внутри триггера мы можем обратиться ко всем полям еще не вставленной записи, используя для этого обращение: New.ID, New.NAME и New.PRICE_1.

Мы можем узнать значение каждого поля вставляемой записи, сравнить его или изменить. Это собственно и делается в этом кусочке кода:

```
IF (NEW.ID IS NULL) THEN
  NEW.ID = GEN_ID(GEN_TABLE_EXAMPLE_ID,1);
```

Сначала в операторе IF...THEN проверяем идентификатор ID на наличие какого-либо значения, ведь он может быть сгенерирован на клиенте. Если значением NEW.ID является NULL, то вызываем функцию GEN_ID, которая увеличивает значение генератора GEN_TABLE_EXAMPLE_ID на единицу и затем возвращает полученное число, которое присваивается полю NEW.ID. Таким образом, мы "на лету" изменили значения во вставляемой записи!

Кроме контекстной переменной NEW, существует ее зеркальный аналог – переменная OLD. В отличие от NEW, OLD содержит старые значения записи, которые удаляются или изменяются. Например, мы можем использовать переменную OLD для получения значений записей, которые удаляются из таблицы:

```
CREATE TRIGGER Table_example_ad0 FOR Table_example
ACTIVE AFTER DELETE POSITION 0
AS
BEGIN
 IF (OLD.id>1000) THEN
   BEGIN
   /*..do something..*/
   OLD.ID=10;
  END
END
```

Здесь мы создаем триггер, который срабатывает ПОСЛЕ УДАЛЕНИЯ (AFTER DELETE). Как видите, мы можем получить доступ к уже удаленным данным. Конечно, присвоение OLD.ID=10; не имеет никакого смысла – присвоенное значение пропадет на выходе из триггера. Однако этот пример показывает, что мы можем перехватить удаляемые значения и записать, например, в некую таблицу, где хранится история всех изменений.

Использование контекстных переменных часто вызывает множество вопросов. Дело в том, что в различных видах триггеров NEW и OLD используются поразному, а в некоторых их вообще невозможно использовать. Если мы рассмотрим триггер в нашем примере, то он вызывается ДО ВСТАВКИ. О каких значениях OLD может идти речь? Ведь вставляется совершенно новая запись! И действительно, контекстная переменная OLD не может быть использована в триггерах BEFORE/AFTER INSERT. А переменная NEW не может быть использована в BEFORE/AFTER DELETE. Обе этих переменные одновременно могут быть использованы в триггерах BEFORE/AFTER UPDATE, причем изменять что-либо можно, только используя переменную NEW (действительно, что можно изменять в удаляющихся значениях, доступных через OLD?), и только в триггерах BEFORE INSERT/UPDATE.

Довольно сложные правила использования, не так ли? Давайте попробуем формализовать их в виде простых правил, которые сведены в таблицу 1.3. В ней мы опишем, как можно применять контекстные переменные в различных триггерах. Эту таблицу удобно использовать в качестве подсказки при разработке триггеров.

В этой таблице для каждой контекстной переменной заведено по два столбца – "Читать" и "Изменять", отражающих возможные действия с этими переменными. В столбце 6 строчек – по числу типов триггеров. Например, если на пересечении типа триггера и возможного действия с контекстной переменной

NEW стоит Y, это значит, что в данном типе триггер можно читать или одновременно читать и менять данные. Если стоит N/A, то в этом триггере нельзя осуществить это действие с данной контекстной переменной.

Таблица 1.3. Использование контекстных переменных NEW и OLD в триггерах

Тип триггера	Контекстные переменные			
	NEW		OLD	
	Читать	Изменять	Читать	Изменять
BEFORE INSERT	Y	Y	N/A	N/A
AFTER INSERT	Y	N/A	N/A	N/A
BEFORE UPDATE	Y	Y	Y	N/A
AFTER UPDATE	Y	N/A	Y	N/A
BEFORE DELETE	N/A	N/A	Y	N/A
AFTER DELETE	N/A	N/A	Y	N/A

Наиболее широкие возможности предоставляет использование NEW и OLD в операции обновления. Ведь таким образом мы можем сравнить текущее (OLD) и новое (NEW) значения и предпринять какие-то действия. Например, такой триггер будет очень полезен для вычисления текущих остатков товара на складе при приходе/расходе товара.

Управление состоянием триггера

По умолчанию триггер создается активным, т. е. он будет срабатывать при осуществлении соответствующей операции. Состоянием триггера управляет ключевое слово ACTIVE в заголовке. Если же триггер сделать неактивным, то он не будет исполняться при возникновении операции. Это бывает полезным при осуществлении каких-либо внеплановых операций над данными, например массовой заливке данных или ручном исправлении данных. Чтобы отключить триггер, необходимо выполнить команду DDL:

```
ALTER TRIGGER <trigger_name> INACTIVE;
```

Обратите внимание, что это команда относится к Data Definition Language, и ее нельзя вызвать из хранимых процедур или других триггеров. Вообще говоря, существует способ управлять состоянием триггеров с помощью модификации системных таблиц. Конечно, модификация системных таблиц является недокументированным способом работы с триггерами и рекомендовать ее мы не будем, но для иллюстрации возможностей работы с системными таблицами InterBase приведем пример. Для того чтобы установить состояние триггера в INACTIVE, достаточно выполнить следующую команду:

```
UPDATE rdb$triggers trg
SET trg.rdb$trigger_inactive=1
WHERE trg.rdb$trigger_name='TABLE_EXAMPLE_AD0'
```

Эта команда аналогична по действию вышеприведенной команде DDL, но ее можно вызывать в других триггерах и процедурах.

Тут следует лишить вас некоторых надежд, которые могли зародиться, когда вы увидели, что метаданные триггеров можно с легкостью изменять с помощью обычного SQL-запроса. Часто такую возможность принимают за хороший способ управлять цепочками триггеров, т. е. в одном триггере или хранимой процедуре включать или выключать нужные триггеры и таким образом управлять обработкой данных, включая или выключая нужные триггеры. Однако изменять состояние триггеров "на лету" не удастся.

Дело в том, что триггеры работают в рамках той же транзакции, что и вызвавшее их изменение. Поэтому если один триггер изменит состояние другого в зависимости от каких-либо условий, то механизм "активных таблиц", который занимается запуском триггеров (хоть мы и говорим, что триггер запускается неявно, но "кто-то внутри сервера" должен их все-таки запускать!), не увидит эти изменения, так как они еще не подтверждены! Таким образом, в рамках одной транзакции нельзя управлять состоянием триггеров.

Если сделать подтверждение транзакции, в которой выполнился первый триггер, который выключил (или включил) второй триггер, а затем запустить снова транзакцию, то мы увидим изменения в состоянии второго триггера. Но какой смысл это делать, ведь суть идеи состояла в том, чтобы включать триггеры на лету, не теряя значения в буфере контекстных переменных NEW или OLD.

В общем, это был пример того, что не следует делать в триггерах. Другим примером того, чего не следует делать в триггерах, является изменение данных в той же таблице, к которой привязан триггер, не через контекстные переменные, а с помощью обычных SQL-команд INSERT/UPDATE/DELETE. Например, некий триггер на вставку вызывает хранимую процедуру, внутри которой происходит вставка записи в ту же таблицу. Вставка опять вызовет срабатывание нашего триггера, и возникнет зацикливание. Следует очень внимательно относиться к использованию триггеров, так как зацикливание в ряде случаев может привести к аварийному завершению сервера InterBase.

Ошибки и исключения в триггерах

Если база достаточно сложная (лучше сказать, достаточно реальная), то вам никак не избежать появления ошибок. Более того, ошибки типа "конфликт с другими пользователями" являются повседневным и нормальным явлением в многопользовательской среде. Как InterBase обрабатывает ошибки в триггерах? Ведь ситуация может быть достаточно нетривиальная – например, вставка записи в главную таблицу запускает хранимую процедуру, которая вставляет записи в подчиненные таблицы, причем при вставке в подчиненные таблицы срабатывают триггеры на вставку, которые получают новые значения генераторов и подставляют их в нужные поля. Можно представить не один подобный уровень вложенности. Что произойдет, когда где-то в "дальних" ветках этого дерева событий возникнет ошибка?

При возникновении ошибок на любом этапе – в триггере, в вызываемых им ХП или в неявно активизируемых других триггерах – InterBase сообщит об ошибке и откатит изменения в таблицах, проведенные в рамках инициировавшего эту це-

почку оператора. Оператор – это предложение INSERT/UPDATE/DELETE или SELECT, а также EXECUTE PROCEDURE.

Таких операторов может быть в транзакции несколько. Отменяется все действия только в рамках оператора, вызвавшего ошибку. Клиентское приложение может отследить возникновение ошибки и подтвердить транзакцию. Другими словами, ошибка в триггере не обязательно требует отката транзакции. Клиентское приложение может обработать ошибку, полученную при выполнении оператора и, например, выполнить вместо этих изменений какие-то другие, если такова логика предметной области, или изменить логику выполнения дальнейших изменений в этой транзакции и подтвердить реально выполненные в транзакции изменения.

Теперь, когда мы знаем, что делает InterBase при возникновении ошибки в триггере, неплохо бы понять, что можем сделать мы, чтобы обработать ошибочную ситуацию. Если мы будем верить в то, что все наши триггеры и ХП не имеют ошибок и конфликтов между действиями пользователей быть не может, то можем вообще не обрабатывать ошибки на уровне базы данных. Если же ошибка возникнет, InterBase пошлет нашему клиентскому приложению сообщение об ошибке, которое мы вольны обработать или нет, – в любом случае InterBase уже выполнил свою миссию – откатил ошибочное действие в триггере. Однако есть и другой путь.

Мы можем воспользоваться обработкой ошибочных ситуаций непосредственно в теле триггера (или хранимой процедуры) с помощью конструкции WHEN...DO. Использование этой конструкции аналогично применению ее в хранимых процедурах, и подробнее об использовании WHEN…DO см. главу "Расширенные возможности языка хранимых процедур InterBase" (ч. 1).

Точно так же как и в хранимых процедурах, в триггерах можно возбуждать собственные исключения. Так как триггер фактически представляет собой разновидность исполнимой хранимой процедуры, то возбуждение в нем исключения прервет работу триггера и приведет к отмене всех действий, совершенных в триггере, – явных и неявных.

События InterBase

Одной из мощных возможностей InterBase, часто используемых в триггерах, являются события (events). События представляют собой строковые сообщения, которые могут быть посланы из триггера или хранимой процедуры. Получат эти события те клиенты InterBase, которые зарегистрированы как заинтересованные в данных событиях. Таким образом, можно оповещать клиента о каких-то изменения внутри базы данных.

События не являются постоянным объектом базы данных – они нигде в базе данных не хранятся, не создаются и не модифицируются, а порождаются "на лету". Чтобы послать какое-то событие, необходимо воспользоваться следующей конструкцией:

```
POST_EVENT 'текст_сообщения';
```

Надо сказать, что 'текст_сообщения' может браться из переменной и, таким образом, можно порождать события динамически, например так:

```
…
If (<какое-то булево выражение>) then
BEGIN
  Event_text ='IT IS TRUE!';
END
ELSE
BEGIN
  Event_text ='IT IS FALSE!';
END
FALSE_EVENT :Event_text;
…
```

Однако если ни одно клиентское приложение, соединенное с базой данных, в которой порождаются какие-то события, не является зарегистрированным на получение этих событий, то все они "уйдут в эфир" и фактически пропадут.

Для регистрации (подписки) на получение нужных событий используют специальные функции InterBase API, которые реализованы, например, в библиотеке FIBPlus – в компоненте SuperIBAlerter.

Как только приложение регистрируется для получения какого-либо события, запись об этом заносится в таблицу блокировок InterBase, которая является единой для всех пользователей сервера InterBase, и сервер начинает просматривать все порождаемые события на предмет появления зарегистрированных данным клиентом. Если такое событие появляется, то клиентское приложение получает соответствующий сигнал, на который может отреагировать каким-либо образом.

События в триггерах являются удобным механизмом для организации протокола изменений в определенных таблицах.

Заключение

Триггеры являются мощным средством для реализации бизнес-логики на стороне сервера. Размещение операций обработки данных в триггерах позволяет упростить и централизовать бизнес-логику приложений, но одновременно несет в себе определенные трудности, связанные с отладкой приложений СУБД на уже работающих базах.

В любом случае при разработке достаточно сложных приложений для СУБД InterBase использование триггеров является одной из возможностей сделать работу создателя СУБД проще и приятнее.

User Defined Functions

Зачастую от программистов, использующих другие серверы баз данных, можно услышать мнение, что SQL InterBase не отличается большим разнообразием встроенных функций. Формально такая точка зрения имеет основания, однако разработчики InterBase сознательно пошли на это ограничение. Как уже неоднократно было сказано, InterBase отличается скромными системными требованиями и занимает мало места на жестком диске. Небольшой совокупный размер файлов продиктован, в частности, тем, что InterBase не перегружен разнообразными дополнительными и, в общем-то, редко используемыми функциями. Зато InterBase включает возможность расширить стандартный набор функций любыми дополнениями, которые нужны в конкретной базе данных. Таким образом, разработчик может реализовать для своих приложений даже такие функции, которые никогда не входят в поставку серверов баз данных.

Механизм подключения функций

Специально для расширения функциональности SQL InterBase предлагает механизм функций, определяемых пользователем (user defined functions). Вы можете создать динамическую библиотеку (Dynamic Link Library) при помощи любой системы разработки, которая позволяет создавать выполнимые файлы данного формата. В частности, можно использовать Borland Delphi, Borland C++ Builder, Microsoft Visual C++ и т. д. Далее, необходимо поместить полученную DLL в каталог, из которого InterBase сможет вызывать библиотеку, и декларировать нужные функции из DLL в своей базе данных при помощи команды DECLARE EXTERNAL FUNCTION. После этого вы сможете вызывать указанные функции, как если бы они были встроенными функциями InterBase.

> InterBase до версии 6.0 требовал, чтобы DLL находилась в любом из каталогов, указанных в системной переменной PATH. InterBase 6 и выше (включая клоны Firebird и Yaffil) требует, чтобы DLL была помещена в специальный каталог UDF, находящийся в общем каталоге установки InterBase.

Создание собственных функций

Мы не будем подробно останавливаться на всех особенностях создания пользовательских функций, поскольку данный механизм достаточно прост, однако продемонстрируем написание и подключение одной функции на примере. В качестве средства разработки мы будем использовать Borland Delphi. Для примера также будет применяться стандартная база данных EMPLOYEE.GDB.

В нашем примере мы создадим функцию, которая будет преобразовывать строку к верхнему регистру. Подобная функция может оказаться полезной, если вы не задали опцию COLLATE для ваших строковых полей. Кроме того, работа со строковыми параметрами, как правило, вызывает наибольшее количество вопросов при создании пользовательских функций. Разумеется, мы исходим из предположения, что вы знакомы с принципом создания DLL при помощи Delphi.

```
library TestUDF;
uses SysUtils;
function malloc(Size: Integer): Pointer; cdecl; external
'msvcrt.dll';

function StrUpperCase(sz: PChar): PChar; cdecl; export;
var Tmp: string;
begin
  Tmp := AnsiUpperCase(sz);
  Result := malloc(length(Tmp) + 1);
  StrPCopy(Result, Tmp);
end;
exports
  StrUpperCase;
begin
end.
```

Динамическая библиотека экспортирует только одну функцию: StrUpperCase. Для передачи строковых параметров, равно как и результата функции, используется тип Pchar, т. е. динамическая строка, ограниченная символами #0. Смысл кода нашей функции очевиден: мы приводим строку sz у верхнему регистру, используя стандартную функцию AnsiUpperCase. Данная функция корректно работает с русскими буквами, если в системе установлена русская кодовая страница. После этого выделяем память для результирующей переменной, используя malloc – стандартную функцию Windows. Остается только скопировать значение временной переменной Tmp в переменную Result. Скомпилируйте библиотеку и поместите полученный файл TestUDF.dll в нужный каталог. Если вы используете InterBase 6.x или его клоны, то это каталог \Udf, который находится в каталоге установки сервера. Необходимо зарегистрировать функцию в базе данных. Для регистрации необходимо выполнить команду DECLARE EXTERNAL FUNCTION, которая имеет следующий синтаксис:

```
DECLARE EXTERNAL FUNCTION name [datatype | CSTRING (int)
[, datatype | CSTRING (int) …]]
RETURNS {datatype [BY VALUE] | CSTRING (int)} [FREE_IT]
ENTRY_POINT 'entryname'
MODULE_NAME 'modulename';
```

Параметр name – это имя пользовательской функции внутри базы данных. Он не обязательно должен совпадать с реальным названием функции в DLL.

Параметр datatype определяет тип параметров. На параметры накладываются следующие ограничения:

- все параметры передаются по ссылке;
- выходной параметр (значение функции) может возвращаться по значению;
- параметры не могут быть массивами.

Если вы хотите применять строковые параметры, то вы должны использовать тип CSTRING. В скобках необходимо указать максимальную длину строки. Если строка является результатом функции, она всегда передается по ссылке, а не по значению.

Параметр FREE_IT указывает InterBase, что после выполнения функции необходимо автоматически освободить память, выделенную для параметров.

Очевидно, данная опция нужна только в том случае, если наша библиотека сама выделила память под какие-либо параметры функции.

В параметре entryname необходимо указывать название функции в DLL, которую мы собственно и хотим декларировать как пользовательскую функцию.

В параметре modulename необходимо указать название файла DLL, в котором находится декларируемая функция пользователя.

Стоит заметить, что параметры entryname и modulename регистрочувствительные.

Таким образом, чтобы добавить нашу функцию в базу данных, нам необходимо выполнить следующую команду:

```
DECLARE EXTERNAL FUNCTION USTRUPPERCASE
    cstring(254)
    RETURNS cstring(254) FREE_IT
    ENTRY_POINT 'StrUpperCase' MODULE_NAME 'TestUDF.dll'
```

После этого мы сможем использовать новую функцию USTRUPPERCASE в любом SQL-запросе. Например, мы можем проверить, как работает функция на следующем запросе:

```
SELECT USTRUPPERCASE(DEPARTMENT) FROM DEPARTMENT
```

Запрос вернет названия отделов из таблицы DEPARTMENT:

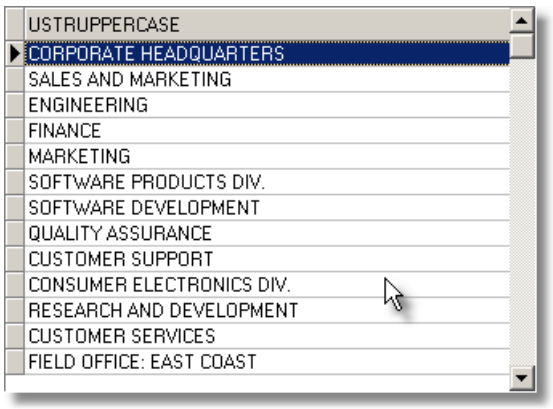


Рис. 1.5. Результат использования UDF USTRUPPERCASE

Заключение

Мы рассмотрели на примере, как можно расширять набор доступных SQL-функций при помощи механизма User Defined Functions. Имея этот простой, но очень мощный механизм, вы сможете сделать обработку бизнес-правил в ваших базах данных гораздо более эффективно и удобно. В сущности, механизм пользовательских функций InterBase имеет только одно серьезное ограничение – он не позволяет обрабатывать NULL-параметры. В остальном функциональность пользовательских "расширений" SQL зависит только от ваших потребностей.

Широкий выбор UDF-библиотек, а также более подробную информацию об их использовании и разработке всегда можно найти на сайтах www.InterBase-world.com и www.ibase.ru.

Русификация InterBase

Как Borland InterBase, так и Firebird – продукты, ориентированные на международного потребителя и позволяющие работать со множеством разных языков, в том числе и русским. Однако по умолчанию InterBase будет ориентироваться на работу с английским языком, поэтому для того, чтобы хранить в базе данных кириллицу и иметь возможность извлекать ее в читабельном виде, необходимо произвести ряд действий по "русификации" базы данных и включению поддержки кириллицы в клиентских приложениях.

Русификация базы данных InterBase

Наборы символов

Чтобы указать InterBase, как интерпретировать и хранить помещаемую в базу данных символьную информацию, необходимо указать *набор символов* (character set), который будет использоваться для представления этих символов в нужном виде (см. выше главу "Типы данных" для информации о хранении символьных данных). Чтобы пользоваться набором символов, необходимо указать его как атрибут объектов в базе данных, а также указать его для использования в клиентском приложении.

Если вы уверены в том, каким набором символов будете пользоваться в вашей базе данных, то можете установить единый набор символов по умолчанию для всей базы данных. Для этого в команде создания базы данных следует указать набор символов по умолчанию с помощью опции DEFAULT CHARACTER SET. Для того чтобы работать с русскими буквами, следует указывать набор символов WIN1251. Можно также использовать набор UNICODE_FSS, который поддерживает любые символы UNICODE в представлении UTF-8. Однако большинство производителей библиотек доступа не полностью поддерживают этот набор символов, поэтому лучше использовать проверенный WIN1251.

Чтобы указать набор символов по умолчанию для всех объектов базы данных, надо применить команду наподобие этой:

```
CREATE DATABASE 'C:\Database\rusbase.gdb' USER 'SYSDBA' PASWORD
'masterkey' DEFAULT CHARACTER SET WIN1251;
```

Указание набора символов по умолчанию не означает, что все таблицы и поля в базе данных должны иметь тот же самый тип. Они будут использовать этот набор символов, если явно не указать какой-нибудь другой. Мы всегда можем переопределить набор символов. Например, мы можем создать таблицу, в которой 3 поля имеют разные наборы символов:

```
CREATE TABLE langTable(
NAME_RUS VARCHAR(255) CHARACTER SET WIN1251,
NAME_ENG VARCHAR(255) CHARACTER SET WIN1250,
NAME_UNI VARCHAR(255) CHARACTER SET UNICODE_FSS);
```

Однако, создав такую таблицу, не думайте, что вы сможете с помощью одного SQL-запроса заносить данные на разных языках в эту таблицу. Дело в том,

что при подключении к базе данных приложение также должно указывать набор символов (см. ниже), при помощи которого оно будет интерпретировать получаемые данные. Поэтому, чтобы записать в каждое поле символы из разных наборов данных, он должен трижды подключиться к базе данных с указанием разных наборов символов либо использовать набор UNICODE_FSS, если клиентская библиотека это позволяет.

Хранение символьных данных без использования наборов символов

Считать данные таблицы, в которой символьные поля имеют разные наборы символа с помощью одного SQL-запроса также не получится, если мы укажем один из трех наборов данных, то InterBase попытается привести все данные к этому набору, а это может у него не получиться – и тогда возникнет ошибка "Cannot transliterate characters between character sets".

Однако существует еще один, недокументированный способ хранить в базе данных символы из различных наборов. Этот способ заключается в том, чтобы вообще не указывать набор символов, тогда по умолчанию будет применять character set NONE, использование которого дает понять InterBase, что символьные данные должны храниться так, как они есть, без всяких интерпретаций. При этом всю ответственность за интерпретацию данных берет на себя разработчик клиентского приложения. Таким образом, отказавшись от использования наборов символов, мы получаем возможность читать и хранить любые символьные данные.

Казалось бы, надо всегда пользоваться CHARACTER SET NONE и избегать различных проблем, однако, отказываясь от наборов символов, мы отказываемся и от полезных свойств, которые они с собой несут. Прежде всего, использование CHARACTER SET, а, точнее, их COLLATION ORDERS (способов упорядочения) позволяет корректно сортировать русские символы и приводить их к верхнему регистру.

Дело в том, что по умолчанию русские буквы сортируются в двоичном порядке, т. е. в соответствии с порядком расположения кодов символов. При этом строчные буквы располагаются после прописных, а буква "Ы" вообще располагается отдельно. Чтобы заставить их сортироваться правильно, надо указать способ упорядочения, т. е. COLLATION ORDER.

Вносим ясность

У начинающих разработчиков часто возникает путаница в голове от многочисленных опций, определяющих поведение InterBase с русскими буквами. Вероятно, дочитав до этого места, вы уже достаточно запутались во множестве взаимнопересекающихся определений. Чтобы внести ясность в эти понятия, давайте еще раз их прокомментируем.

Основа всего – символьные типы InterBase, которые позволяют хранить в каждом поле до 32767 байт. Однако байты в общем случае не равнозначны символам, потому что в различных системах кодировки для представления одного символа могут использоваться 1, 2 или даже 3 байта. Таким образом, когда мы определяем поле типа CHAR или VARCHAR, то мы задаем количество симво-

лов, которые там могут поместиться, а количество байтов определяется умножением заданной длины на максимальный размер символа для данного набора символов. Для набора символов WIN1251 любой символ занимает 1 байт, поэтому размер поля в байтах будет равен объявленной длине. А вот при использовании кодировки UNICODE_FSS максимальный размер символа составляет 3 байта; таким образом, реальная длина поля будет равна утроенному объявленному размеру. Вы не сможете создать поле с кодировкой UNICODE_FSS с длиной более чем 32767 div 3 = 10921 символ. В то же время реальный размер символа может быть меньше максимального, использовавшегося InterBase для расчета размера поля, т. е. вы сможете записать в такое поле больше символов, чем его объявленная длина!

Наборы символов (CHARACTER SET) – это фактически таблицы перекодировки физического представления (где один символ занимает 2 или 3 байта) в желаемое (т. е. такое, каким эти символы хотят видеть соответствующие клиентские приложения).

В InterBase существует множество наборов символов, полный список которых можно найти в документации по InterBase. Каждый набор символов использует для хранения тех или иных символов разное количество байт. Проще говоря, это таблица, где каждому символу поставлен в соответствие 1- или 2- или 3-байтовый код.

Когда мы создаем символьное поле и указываем набор символов явно или с помощью установок по умолчанию, то в этом случае мы неявно задаем, сколько символов поместится в это поле. Например, набор символов UNICODE_FSS использует 2 байта для кодирования русских букв. Следовательно, мы можем поместить в поле, объявленное как VARCHAR(255) CHARACTER SET UNICODE_FSS, количество русских символов, равное 255 * 3 div 2, т. е. 382.

Далее, каждый набор символов (CHARACTER SET) имеет свой порядок сортировки по умолчанию. Очень часто этот порядок сортировки не отражает принятую в языке сортировку конкретных символов. Например, русский набор символов WIN1251 неправильно сортирует символы русского алфавита, т. е. сортирует их в порядке следования двоичных кодов символов.

Как же изменить сортировку символов по умолчанию внутри набора символов? Для этого применяются порядки (или способы) сортировки наборов символов – так называемые COLLATION ORDERS. Дополнительно с каждым COLLATION ORDER связаны таблицы преобразования в нижний и верхний регистр. Для каждого набора символов существуют свои определенные порядки сортировки. Например, для самого распространенного русского набора символов WIN1251 существуют два способа сортировки: WIN1251, который задается по умолчанию, и опциональная сортировка PWX_CYRL. В порядке сортировки PXW_CYRL одни и те же прописные и строчные буквы имеют одинаковый вес, т. е. этот порядок сортировки не зависит от регистра символов (case insensitive). Русские буквы будут располагаться в следующем порядке: аАбБвВ...яЯ.

Как устроены способы сортировки? В COLLATION ORDERS строится дополнительная таблица пересортировки, определяющая порядок (условный вес) символов при сортировке. И в этой таблице для представления порядка символа может использоваться несколько байтов (2 или 3, например). В частности,

в COLLATION ORDER WIN1251 используется 1 байт для представления символа и его порядка, а в опциональной PWX_CYRL – целых 3 байта!

Размер представления веса символа в таблице порядка сортировки имеет значение при использовании индексов по полям символьных типов данных. Дело в то, что в индексе хранятся не исходные символьные строки, а ключи, полученные из строки на основе таблицы сортировки. Размер ключа может быть больше размера исходной строки: так, для порядка сортировки PXW_CYRL размер ключа может быть больше максимум в 3 раза, чем исходная строка. InterBase использует максимальный коэффициент для ограничения размера индекса по символьным полям. Таким образом, при максимальном размере индекса 254 байта вы не сможете создать индекс по полю с длиной более 254 div 3 = 84 символа. Поэтому использование COLLATION ORDER может оказаться вещью, требующей достаточно многих ресурсов. Тем не менее весь вопрос в том, как ее применять. Параметр COLLATE можно использовать и по требованию, прямо в тексте запроса, не указывая его в самом определении поля. Когда мы указываем в запросе использовать для сортировки или приведения к верхнему регистру какой-либо способ сортировки, то InterBase сам достраивает данные из этого поля с учетом указанного способа.

Использовать конкретный COLLATE в запросе очень легко, правда, в этом случае не может быть использован индекс:

```
SELECT *
FROM table1
ORDER BY SYMBOLIC_FIELD1 COLLATE PXW_CYRL
```

Другой пример – для встроенной функции UPPER:

```
SELECT UPPER(field1 COLLATE PXW_CYRL) From table1
```

Этот способ является наиболее гибким: хранить можно лишь сами символы, а специальный порядок сортировки применять, только когда это требуется.

Русификация клиентских приложений InterBase

Разобравшись с русификацией баз данных InterBase и с тем, как обеспечить хранение и интерпретацию символов на уровне базы данных, необходимо рассмотреть вопрос о том, как сделать так, чтобы клиентские приложения могли корректно читать и записывать символьные данные.

Принцип здесь очень простой: при подключении к базе данных в клиентских приложениях необходимо указать точно такой же набор символов, как и в самой базе данных. В этом случае передаваемые данные будут интерпретироваться правильно и появляться у пользователя на экране в виде корректных символов, а не бессмысленных значков.

Способ указания того, какой набор символов использовать, различен для каждой библиотеки доступа. Для библиотек FIBPlus и IBX для подключения к базе данных с набором символов, например, WIN1251, нужно указать в параметрах подключения lc_type=WIN1251.

Для работы с базами данных, использующими WIN1251, через Borland Database Engine необходимо указать в параметрах псевдонима LANGDRIVER=PDOX ANSI Cyrr.

Для работы с базой данных через JDBC необходимо указать в настройках драйвера строку charset=cp1251 (подробнее см. в главе "Разработка клиента InterBase на Java" (ч. 3)).

Собственные наборы символов и способы сортировки

Сами кодировки хранятся в файле gdsintl.dll, который находится в каталоге %INTERBASE%\Intl. Вы можете самостоятельно разрабатывать и подключать свои собственные наборы символов и COLATION ORDERS в InterBase и во все его клоны. Для их разработки существует специальный инструментарий, ссылки на который вы можете найти на сайте www.InterBase-world.com.

Транзакции. Параметры транзакций

Концепция транзакций

Что такое транзакции?

В этой книге практически в каждой главе упоминаются транзакции. Понятие транзакции пронизывает всю теорию и практику работы с базами данных. Транзакции всегда, транзакции везде – вот лозунг разработчиков СУБД.

Понятие транзакции само по себе чрезвычайно простое и очевидное.

> Транзакция – это логический блок, объединяющий одну или более операций в базе данных и позволяющий подтвердить или отменить результаты работы всех операций в блоке.

Возможность отмены – только одно из свойств. Определение обычно дается очень обтекаемое: транзакция – последовательность операций с базой данных, логически выполняемая как единое целое. Транзакция обладает свойствами атомарности, согласованности, изоляции и долговременности (по-английски ACID – Atomicity, Consistency, Isolation, Durability).

Давайте рассмотрим более подробно это определение. Операции, о которых идет речь в определении – это INSERT/UPDATE/DELETE и, конечно, SELECT. Если транзакция объединяет какие-то операции в единый блок, то говорят, что эти действия выполняются в *контексте* данной транзакции.

Вторая часть определения гласит: – "Позволяющий подтвердить или отменить результаты работы всех операций в блоке". Это очень важная часть, содержащая в себе суть идеи транзакций. Эту часть определения мы проиллюстрируем нижеследующим классическим примером транзакции.

Представьте себе перевод денег с одного счета в банке на другой. Когда клиент инициирует перевод денег, то начинается транзакция. Деньги снимаются со счета-источника и переводятся на счет-приемник. Когда приходит подтверждение, что деньги переведены, транзакция завершается, т. е. именно в этот момент происходит "узаконивание" перевода денег. Если же хотя бы один этап перевода не состоялся, то транзакция откатывается и все проведенные в ее рамках изменения отменяются. Только после подтверждения транзакции пришедшие на счет деньги станут "реальными" (а ушедшие со счета – реально исчезнут).

Понятие транзакции как логического блока операций, которым можно оперировать как единым целым при подтверждении/отмене результатов, очень популярно среди программистов-разработчиков баз данных. Оно позволяет объяснить большинство феноменов, которые происходят в приложениях баз данных и в то же время достаточно простое для интуитивного понимания.

В сущности, это определение абсолютно верно для всех случаев жизни, но в то же время оно не подчеркивает некоторых важных особенностей транзакций, которые могут проявиться в самый неподходящий момент и удивить даже опытного разработчика. Чтобы разобраться в сущности транзакций в InterBase, нам придется пойти дальше этого простого определения и углубиться в тонкости.

Прежде всего давайте внесем важное уточнение в определение транзакции:

> Транзакция – это механизм, позволяющий объединять различные действия в логические блоки и обеспечить возможность принимать решения об успешности действий всего блока операций в целом.

Обратите внимание на смещение акцента в определении: "транзакция – это механизм…"! Именно от представления о транзакции как о механизме, выполняющем определенные функции, мы и будем отталкиваться в дальнейших рассуждениях.

Давайте разберемся в некоторых фактах, которые необходимо знать для того, чтобы двигаться дальше в понимании транзакций.

1. Механизм транзакций обязательно используется для ВСЕХ операций в базе данных (о некоторых особых случаях будет рассказано ниже). Возможно, некоторые разработчики, пользующиеся высокоуровневыми инструментами разработки приложений баз данных, могут заявить, что они никогда не применяли транзакции и не видят в них нужды. Но это всего лишь означает, что всю работу по управлению транзакциями брал на себя инструмент разработки (и вряд ли он управлял ими достаточно эффективно!).
2. Сочетание слов "логический блок" напоминает нам, что транзакции изначально задумывались и реализовывались как механизм управления бизнес-логикой в базах данных. Это означает, что объединением некоторой последовательности операций в транзакцию управляет клиентское приложение базы данных (а в конечном итоге – пользователь).
3. Подтверждение или отмена результатов операций, объединенных одной транзакцией, не означает, что все эти операции выполнились успешно (или закончились ошибкой). Подтверждение транзакции – это решение о том, что следует оставить в базе данных результаты работы всех операций, входящих в транзакцию, вне зависимости от того, как они закончились. Если клиентское приложение (фактически человек-пользователь) решило подтвердить результаты транзакции, то при этом подтвердятся результаты всех успешных действий (а у неуспешных действий просто не будет результатов, поэтому произойдет лишь формальное подтверждение). Если клиентское приложение решило откатить транзакцию, то все результаты всех действий и успешные, и неуспешные, будут аннулированы.

Хочется отметить, что начинающие пользователи часто ставят знак равенства между подтверждением транзакции и успешностью проведенных в ее рамках действий. На самом деле здесь нет четкой связи – решение о подтверждении транзакции принимается на основании логики клиентского приложения (проще говоря, зависит от произвола пользователя). Да, обычно если все операции в транзакции прошли успешно, то транзакция подтверждается и все полученные результаты "узакониваются", но такое поведение не является обязательным. Ведь ничто не должно мешать пользователю отменить результаты успешных операций, исходя из каких-то своих высших соображений.

Пора разобраться в понятиях подтверждения ("узаконивания") и отмены (отката) транзакций. Следующий раздел внесет ясность в этот вопрос.

Изолированность транзакций

Давайте углубимся в рассмотрение того, зачем нужны транзакции в базе данных. Пример с переводом денег дает верную подсказку, представляя транзакцию как некоторый черный ящик, в котором производятся действия над содержимым базы данных. В этот ящик нельзя заглянуть до того, как транзакция завершится подтверждением или откатом. Если бы сумели все-таки заглянуть "внутрь" транзакции, в контексте которой осуществляется перевод денег, то увидели бы печальную картину, что-то вроде того, что: деньги уже ушли с одного счета, но на другой не пришли, или, наоборот, пришли, но "размножились" и существуют сразу на обоих счетах. Другими словами, "внутри" транзакции база данных в какие-то моменты находится в неправильном с точки зрения бизнес-логики состоянии. Такое неправильное состояние называется "нецелостным", а правильное соответственно – "целостным".

> Целостное состояние базы данных – состояние, в котором база данных содержит корректную информацию, соответствующую правилам бизнес-логики, применяющимся в данном приложении.

Отсюда следует еще одно определение транзакции:

> Транзакция – механизм, позволяющий переводить базу данных из одного целостного состояния в другое.

Обратите внимание на слово "позволяющий": оно подчеркивает потенциальность возможностей механизма транзакций по обеспечению целостности базы данных.

Во время выполнения транзакции результаты операций, выполняющихся в ее контексте, невидимы для остальных пользователей, вплоть до момента подтверждения. Учитывая то, что все действия выполняются в контексте транзакций, то можно утверждать следующее: *результаты операций, выполняющихся в контексте одной транзакции, невидимы для операций, осуществляющихся в контексте других транзакций.*

> Конечно, это только в идеальном случае, когда транзакции полностью изолированы. В реальности добиться такого трудно из-за сокращения числа одновременно выполняемых транзакций, работающих с общими данными. Поэтому для каждой транзакции предусматривается уровень изоляции, выбираемый как компромисс между снижением производительности и допустимым нарушением изолированности.

Давайте поясним эти мысли с помощью такого примера.

Предположим, у нас есть документ, разные части которого хранятся в нескольких таблицах – таблице заголовка и нескольких таблицах-подробностях. Очевидно, что документ должен существовать только как целостная сущность, у которой заполнены параметры заголовка и корректно определены данные в таблицах-подробностях. Также очевидно, что процесс составления документа может быть достаточно длительным – сначала пользователь введет заголовок, затем заполнит содержание и т. д. В то же время, другие пользователи должны увидеть документ в целостном виде, т. е. не может быть документа без заголовка или с некорректными данными.

В данном примере транзакция начинается в момент начала создания/редактирования документа и заканчивается после окончания этого редактирования, т. е. тогда, когда пользователь, редактирующий документ, сочтет, что документ находится в целостном состоянии.

Следует четко понимать отличия двух понятий – *изолированности*, когда одна транзакция не видит изменения, совершаемые в контексте другой (причем это настраивается уровнями изоляции), и *целостности*, когда состояние всей базы (а не с точки зрения какой-то отдельной транзакции) после завершения транзакции всегда должно соответствовать всем ограничениям предметной области (правилам бизнес-логики). В течение выполнения транзакции в принципе возможны нарушения целостности.

В результате все остальные пользователи в базе данных видят документ только в целостном состоянии, соответствующем правилам бизнес-логики для данной задачи.

Таким образом, можно сформулировать наиболее общее определение механизма транзакций:

> Транзакция – это механизм, позволяющий объединять различные действия в логические блоки и обеспечить возможность принимать решения об успешности действий всего блока операций в целом. Логические блоки операций осуществляют перевод базы данных из одного целостного состояния, соответствующего бизнес-правилам задачи, в другое целостное состояние. Механизм транзакций служит для обеспечения изоляции изменений, совершаемых операциями в контексте одной транзакции, от операций в других транзакциях.

Можно считать, что это определение достаточно точно отражает понятие транзакции. Это определение применимо для всех реляционных СУБД. Однако предметом рассмотрения данной книги является InterBase, поэтому в следующих разделах мы будем рассматривать конкретные особенности механизма транзакций именно в InterBase.

Механизм транзакций в InterBase

Надо сказать, что реализация транзакций в InterBase отличается от реализации транзакции в большинстве других СУБД. Это связано с особой архитектурой баз данных InterBase, именуемой Multi Generation Architecture (MGA) – многоверсионной архитектурой.

Чтобы разобраться в реализации транзакций в InterBase, придется совершить экскурс в многоверсионную архитектуру баз данных, а также затронуть аспекты низкоуровневой реализации ядра InterBase. Возможно, в процессе чтения этого раздела необходимо будет обратиться к главе "Структура базы данных InterBase".

Итак, приступим. Давайте сначала разберем сущность многоверсионной архитектуры.

Многоверсионная архитектура InterBase

InterBase – это первая в мире СУБД, в которой реализована многоверсионная архитектура. Именно многоверсионная архитектура позволяет организовать взаимодействие пользователей таким образом, что читающие пользователи не блокируют пишущих, а также дает возможность очень быстро восстанавли-

ваться после сбоев в базе данных и отказаться от ведения протокола транзакций (transaction log), а также предоставляет массу других преимуществ.

Сущность многоверсионной архитектуры достаточно проста. Основная идея состоит в том, что все изменения, проводимые над конкретными записями (а к этому сводятся любые операции над информацией в базе данных), производятся не над самой записью, а над ее версией. *Версия записи* – это копия записи, которая создается при попытке ее изменить.

Можно также сказать, что для каждой записи возможно существование нескольких ее версий, при этом каждая транзакция видит только одну их этих версий.

Пусть у нас есть некоторое начальное состояние базы данных, в котором имеется таблица с одной записью. Для простоты предположим, что сначала нет подключенных к базе данных пользователей и соответственно нет никаких изменений в данных. Когда к базе данных подключится пользователь и запустит транзакцию, в рамках которой он начнет производить какие-нибудь изменения над этой записью, то специально для этого пользователя (точнее, для транзакции, в контексте которой производятся операции) запись, содержащаяся в таблице, будет скопирована – появится версия записи. Эта версия целиком принадлежит транзакции, и все операции в рамках этой транзакции будут производить изменения над *версией* записи, а не над исходным оригиналом.

Далее, транзакция может либо подтвердиться, либо отмениться. При подтверждении транзакции произойдет следующее: InterBase попытается пометить предыдущую (исходную) версию записи как удаленную и сделать текущую (измененную в рамках этой завершающейся транзакции) версию основной. Если только один пользователь менял запись, то именно так все и произойдет – измененная версия записи станет основной и именно ее увидят все остальные операции в транзакциях, которые запустятся позже подтверждения.

Предположим теперь, что после запуска описанной в примере транзакции (назовем ее № 1), но до подтверждения ее результатов запустится транзакция № 2, в которой пользователь желает прочитать изменяющуюся запись. Так как неподтвержденные в № 1 изменения нельзя увидеть (в том числе и в транзакции № 2) по крайней мере до подтверждения этой транзакции, то транзакция № 2 увидит предыдущую версию записи!

Как видите, идея версионности гениально проста – дать каждой выполняющейся транзакции по своей собственной версии записей и пусть они наслаждаются одновременной работой с данными – читающие пользователи не мешают пишущему пользователю.

Но обратите внимание, что пишущий пользователь всегда может быть только один! Негоже давать изменять запись сразу двоим пользователям. Если попытаться редактировать одну и ту же запись одновременно в разных транзакциях, то, в зависимости от параметров транзакции, возникнет конфликт обновления записей – в той или иной форме. См. ниже раздел "Конфликты в транзакциях".

Итак, основной постулат многоверсионной архитектуры изложен. Конечно, в случае одновременной работы множества пользователей могут возникать сложные комбинации версий, и это может привести к странным на первый взгляд конфликтам.

Чтобы сформировать четкую и ясную картину того, как работает многоверсионность данных и транзакции в InterBase, придется углубиться в их реализацию на уровне базы данных InterBase.

Реализация многоверсионности. Страницы учета транзакций

Каждая транзакция в InterBase имеет свой уникальный идентификатор – transaction ID или TID (фактически это номер транзакции с момента создания базы данных или последнего restore). Каждая транзакция, запускаясь, получает свой номер, который последовательно увеличивается – т. е. более старые транзакции имеют меньшие номера, чем новые.

Для учета транзакций используются страницы учета транзакций (Transaction Inventory Page, TIP). Когда в InterBase начинается транзакция, то на странице учета транзакций появляется отметка о том, что транзакция с определенным идентификатором TID# находится в состоянии выполнения (т. е. она является активной – active).

Всего имеется 4 возможных состояния транзакции: активная (active), подтвержденная (Commited), отмененная (Rolled back) и лимбо (limbo). Статус активной имеют выполняющиеся в данный момент транзакции.

Подтвержденные транзакции – обычно те транзакции, что завершились командой Commit.

Отмененные транзакции – это обычно те транзакции, которые завершились командой Rollback.

Транзакции Лимбо – это транзакции с неопределенным статусом, которые возникают при использовании механизма двухфазного подтверждения транзакций, который применяется при проведении транзакций сразу в нескольких базах данных.

Когда операция в контексте транзакции с идентификатором TID# изменяет какие-либо записи, то для этого изменения создаются версии записей. И каждая версия помечается идентификатором TID# – на физическом уровне это выглядит как номер транзакции в заголовке версии записи.

Если транзакция с номером TID# подтверждается, т. е. переходит в состояние Commited (а процесс подтверждения часто называют commit), то на странице учета транзакций производится отметка об этом. При этом никаких действий над измененными записями не происходит!

Но как же следующие транзакции узнают о том, какая версия записи является актуальной – та ли, которая была изменена транзакцией TID#, или исходная, помеченная другой транзакцией. Вот здесь заключается один фокус – читающая транзакция считывает все версии измененной записи, берет номера транзакций из заголовков записей и та запись считается актуальной, чей номер больше.

При чтении версий записей происходит дополнительная проверка и на подтвержденность транзакции, которая создала версию. Для этого в момент чтения версий не просто сравниваются номера транзакций, чтобы определить актуальную запись, но еще и проверяется на странице учета транзакций, а в каком состоянии находится транзакция с определенным TID.

Если транзакция, создавшая версию записи, откачена (т. е. находится в состоянии Rollbacked), то такая версия является *мусором* (*garbage*) и ее необходимо удалить.

Если же транзакция, создавшая версию записи, подтверждена (находится в состоянии Committed), то такую запись можно считать полноценной претенденткой на самую актуальную запись.

"Почему претенденткой?" – может спросить уважаемый читатель. А потому что может быть две и больше версий записей, которые созданы подтвержденными на текущий момент транзакциями. И поэтому читающая запись выберет среди этих версий записей в качестве актуальной ту версию, в которой TID# больше.

Сборка мусора

Как видите, при использовании многоверсионной архитектуры постоянно накапливаются устаревшие версии, называемые "*мусором"*. Эти версии не являются актуальными и подлежат удалению. Процесс удаления ненужных версий записей называется *сборкой "мусора"*..

Главное, что следует отметить в сборке "*мусора"*.– это то, что она является кооперативной. Как вы поняли из предыдущего описания, транзакции, изменяющие данные, не "убирают за собой": когда происходит завершение, то на странице учета транзакций просто ставится отметка о том, что транзакция с определенным TID подтверждена (committed). При этом не происходит удаления старых версий записей.

Сборка мусора происходит, когда какая-либо транзакция пожелает прочитать данную запись. Эта транзакция считывает все существующие версии этой записи, выясняет по таблице TIP, что версии устарели и удаляет их.

Взаимодействие транзакций

Интересен процесс определения, является ли текущая версия мусором или, возможно, она еще нужна какой-то транзакции.

Для описания этого процесса придется ввести несколько важных понятий. Прежде всего, надо отметить, что все определения строятся относительно какой-либо транзакции, которую называют *текущей*, – обычно это та транзакция, поведение которой необходимо исследовать и которой надо принять решение, что является мусором, а что нет.

Итак, определения:

Заинтересованная транзакция – это транзакция, конкурирующая с текущей.

Старейшая заинтересованная транзакция (oldest interesting transaction) – это старейшая транзакция, конкурирующая с текущей транзакцией.

Каждая транзакция (и текущая тоже, разумеется) имеет *"маску транзакций"*, которая представляет собой снимок страницы учета транзакций, начиная от старейшей заинтересованной транзакции до текущей.

Старейшая активная транзакция (oldest active transaction, OAT) – это транзакция, которая была активной в тот момент, когда запускалась самая старая из активных транзакций в момент запуска текущей.

Именно старейшая активная транзакция и занимается сборкой мусора, так как все остальные транзакции и их изменения "моложе" ее.

Обратите внимание на два момента: во-первых, старейшая активная транзакция – это не постоянно существующая транзакция, а всего лишь обязанность, которую получают транзакции; во-вторых, старейшая транзакция убирает только мусор от завершившихся транзакций, которые еще старше ее! Другими словами, следует рассматривать процесс сборки мусора динамически – как постоянную передачу обязанностей по сборке мусора от одной транзакции к другой.

Разумеется, здесь приведено лишь краткое изложение вопросов, связанных с многоверсионной архитектурой InterBase и ее особенностями. На сайтах www.InterBase-world.com и www.ibase.ru читатель сможет ознакомиться с множеством статей по данной проблеме.

Уровни изоляции транзакций

Как уже было сказано выше, транзакции обеспечивают изолирование проводящихся в их контексте изменений, так что эти изменения невидимы пользователям вплоть до подтверждения транзакции. Но вот вопрос: а должна ли транзакция видеть те изменения, которые были подтверждены другими транзакциями уже после ее запуска? Вот пример.

Допустим, у нас есть таблица с данными, к которой обращаются два пользователя одновременно – один из них изменяет данные, а второй читает. Возникает вопрос, должен ли (или может ли) пользователь, читающий таблицу, видеть изменения, производимые другим пользователем.

Должен или нет – это определяется уровнем изоляции.

> Уровень изолированности транзакции определяет, какие изменения, сделанные в других транзакциях, будут видны в данной транзакции.

Каждая транзакция имеет свой уровень изоляции, который устанавливается при ее запуске и остается неизменным в течение всей ее жизни.

Транзакции в InterBase могут иметь 3 основных возможных уровня изоляции: READ COMMITTED, SNAPSHOT и SNAPSHOT TABLE STABILITY. Каждый из этих трех уровней изоляции определяет правила видимости тех действий, которые выполняются другими транзакциями. Давайте рассмотрим уровни изоляции более подробно.

- READ COMMITTED. Буквально переводится как "читать подтвержденные данные", однако это не совсем (точнее, не всегда) так. Уровень изоляции READ COMMITTED используется, когда мы желаем видеть все подтвержденные результаты параллельно выполняющихся (т. е. в рамках других транзакций) действий. Этот уровень изоляции гарантирует, что мы НЕ сможем прочитать неподтвержденные данные, измененные в других транзакциях, и делает ВОЗМОЖНЫМ прочитать подтвержденные данные.
- SNAPSHOT. Этот уровень изоляции используется для создания "моментального" снимка базы данных. Все операции чтения данных, выполняемые в рамках транзакции с уровнем изоляции SNAPSHOT, будут видеть только состояние базы данных на момент начала запуска транзакции. Все изменения, сделанные в параллельных подтвержденных (и разумеется, неподтвержден-

ных) транзакциях, не видны в этой транзакции. В то же время SNAPSHOT не блокирует данные, которые он не изменяет.

- SNAPSHOT TABLE STABILITY. Это уровень изоляции также создает "моментальный" снимок базы данных, но одновременно блокирует на запись данные, задействованные в операциях, выполняемые данной транзакцией. Это означает, что если транзакция SNAPSHOT TABLE STABILITY изменила данные в какой-нибудь таблице, то после этого данные в этой таблице уже не могут быть изменены в других параллельных транзакциях. Кроме того, транзакции с уровнем изоляции SNAPSHOT TABLE STABILITY не могут получить доступ к таблице, если данные в них уже изменяются в контексте других транзакций.

Параметры транзакций

В первом разделе этой главы была сделана попытка рассмотреть механизм работы транзакций в СУБД InterBase в целом. Теперь необходимо рассмотреть практические аспекты применяющие транзакций в InterBase.

Программисты, использующие такие современные библиотеки для доступа к базам данных InterBase, как FIBPlus, IBProvider, IBX и IBObjects (см. главу "Обзор библиотек доступа к InterBase"), имеют возможность гибко управлять параметрами транзакций для получения наилучших результатов. Поэтому имеет смысл рассматривать параметры транзакций именно в интерпретации для этих библиотек.

Настройка параметров транзакции осуществляется с помощью перечисления набора констант, определяющих поведение транзакции, например, уровень изоляции. Эти константы пришли из InterBase API и имеют следующий вид: isc_tpb_read, isc_tpb_write, isc_tpb_ read_committed и т. д.

Обычно префикс isc_tpb_ опускается и константы для определения параметров транзакции пишутся без него.

Давайте рассмотрим значение и синтаксис применения каждой константы.

Виды параметров транзакции

Все параметры транзакции можно подразделить на группы, каждая из которых отвечает за определенный момент в поведении транзакций. Эти группы приведены в таблице 1.4:

Таблица 1.4. Параметры транзакций

Группы параметров	Константа	Краткое описание константы
Режим доступа	Read	Разрешает только операции чтения
	write	Разрешает операции записи
Режим блокировки	Wait	Устанавливает режим отсроченного разрешения конфликтов. См. ниже раздел "Режим блокировки"
	nowait	При возникновении конфликта немедленно возникает ошибка

Группы параметров	Константа	Краткое описание константы
Уровень изоляции	read_committed rec_version	Возможность читать подтвержденные данные других транзакций. Дополнительный параметр rec_version позволяет читать записи, имеющие неподтвержденные версии
	read_committed no_rec_version	Возможность читать подтвержденные данные других транзакций. Дополнительный параметр no_rec_version не позволяет читать записи, имеющие неподтвержденные версии
	concurrency	При запуске транзакции создается мгновенный "снимок" состояния базы данных (точнее, копируется "маска транзакций" на этот момент), поэтому изменения, сделанные в других транзакциях, не видны в этой транзакции
	consistency	Аналогичен уровню concurrency, но помимо этого блокирует таблицу на запись. См. ниже

Обилие параметров впечатляет, ведь их сочетания должны покрывать все возможные нужды разработчиков приложений баз данных. Однако обычно используется лишь небольшой набор параметров для определения необходимых видов транзакций. Давайте подробно рассмотрим каждую группу параметров транзакций.

Режим доступа

Режим доступа определяет, какие операции могут осуществляться в контексте транзакции. По умолчанию (т. е. если ничего не указывать) ставится режим чтения-записи, т. е. могут осуществляться любые операции. Часто задают вопрос, имеет ли смысл запускать транзакции в режиме только для чтения, если не предполагается операций по изменению данных. Ответ: да, имеет. Особенно в последних версиях InterBase (начиная с 6.5) и Firebird. Транзакции с режимом доступа только для чтения меньше нагружают сервер, так как не создают лишних версий записей.

Для установки режима чтения-записи используется сочетание констант read write. Обычно константы записывают в столбик одну под другой, вот так:

```
read
write
```

Режим блокировки

Режим блокировки определяет, как будут разрешаться конфликты. Если возникает конфликт, то у транзакции, обнаружившей конфликт, есть два выхода –

либо немедленно возбудить исключение, либо подождать некоторое время, после чего опять попытаться разрешить конфликт.

Соответственно есть два варианта режима блокировки – wait и nowait. По умолчанию используется режим wait.

Конфликты, о которых идет речь, возникают как в случае чтения записей, так и в случае записи. На конфликты при чтении записей, помимо wait/nowait, влияют также установки уровня изоляции, и поэтому мы их рассмотрим в разделе про уровни изоляции. А вот на объяснение влияния режима блокировки на конфликты при записи уровень изоляции не влияет, поэтому мы сейчас рассмотрим его.

1. Рассмотрим случай, когда транзакция А *вставляет* запись, но еще не подтвердила ее. Затем в рамках другой транзакции, Б, делается попытка вставить запись с тем же самым первичным ключом, уже вставлена в транзакции А. Вот здесь и начинаются отличия между режимами блокировки:
   - если транзакция Б запущена в режиме wait, то она будет ожидать завершения транзакции А, и если А завершится подтверждением (commit), то вставленная в Б запись будет признана неактуальной и возникнет ошибка Deadlock, а если она откатится (rollback), то изменения в транзакции Б будут приняты и она сможет подтвердить их (т. е. сделать commit);
   - если транзакция Б запущена в nowait, то немедленно возникнет ошибка 'lock conflict on no wait transaction'.
2. Рассмотрим другой случай: транзакция А *изменила* запись, но еще не подтвердила ее изменения. Транзакция Б пытается удалить или изменять эту же самую запись. Опять влияет режим блокировки:
   - если Б в режиме wait, то она будет ждать пока А не подтвердится или не отменится; если А подтвердится, то в Б возникнет ошибка 'Deadlock – update conflict with concurrent update', – потому как А подтвердила свои изменения, изменения в Б признаются неактуальными; если же транзакция А откатится, Б получит возможность подтвердиться;
   - если Б в режиме nowait, то немедленно возникнет ошибка 'lock conflict on no wait transaction'.

Возможно, неясна практическая ценность описаний режима блокировки. Однако чуть ниже это поможет понять суть рекомендованных наборов параметров для транзакций в типичных приложениях баз данных.

Вы, вероятно, заметили, что в сообщении об ошибке конфликта блокировки фигурирует слово "deadlock", однако это слово выбрано не совсем удачно. В переводе с английского оно означает "мертвая блокировка", или "взаимоблокировка". В нашем случае, несмотря на грозное сообщение, никаких взаимоблокировок не возникает.

Что же такое взаимоблокировка на самом деле и когда она может возникнуть?

Взаимоблокировка

Взаимоблокировка – классическая проблема при синхронизации доступа к ресурсу, при котором принципиально невозможна дальнейшая работа конкурирующих транзакций. Для иллюстрации рассмотрим две транзакции T1 и T2 и два ресурса – A и B; в контексте разговора о базах данных ресурсами могут быть, например, записи в некоторой таблице. Допустим, выполняется такая последовательность действий:

1. Транзакция T1 блокирует ресурс A, после чего благополучно работает с ним.
2. Транзакция T2 блокирует ресурс B, после чего также с ним работает.
3. Транзакция T1 желает поработать с ресурсом B, для чего она пытается установить на него блокировку. Так как в это время ресурс B уже занят транзакцией B, транзакция T1 входит в состояние ожидания.
4. Транзакция T2 желает поработать с ресурсом A, пытается выполнить его блокирование и также переходит в состояние ожидания.

В результате мы имеем печальную ситуацию: транзакции не имеют никакого шанса продолжить свое выполнение из-за того, что намертво блокируют друг друга. Любой сервер баз данных должен быть способен выходить из этой ситуации по возможности достойно, и InterBase не исключение. Проблема решается выбором одной из транзакций в качестве жертвы и ее откате, при этом другая транзакция получает возможность выполниться до конца.

Алгоритм распознавания ситуации взаимоблокировки в InterBase не запускается сразу при возникновении конфликта, что сделано из соображений производительности. Вместо этого выдерживается определенный интервал времени, задаваемый параметром DEADLOCK_TIMEOUT в конфигурационном файле InterBase ibconfig, только после этого и производится сканирование таблицы блокировок на предмет взаимного блокирования.

В реальной практике программирования баз данных взаимоблокировки возникают крайне редко, поэтому не стоит считать, что вам так повезло, увидев слово "deadlock" в сообщении об ошибке. Скорее всего это всего лишь конфликт обновления.

Теперь давайте рассмотрим параметры, которые влияют на уровни изоляции.

Установка уровней изоляции

Итак, как было упомянуто выше, уровень изоляции транзакции определяет, какие изменения, сделанные в других транзакциях, может видеть данная транзакция.

Как было сказано в разделе "Уровни изоляции", в InterBase есть 3 основных уровня изоляции. Теоретически существует также четвертый уровень изоляции, так называемое DIRTY READ – "грязное чтение". Транзакции с уровнем изоляции DIRTY READ могут читать неподтвержденные данные в других транзакциях. В InterBase пользователю нельзя запускать транзакции с таким уровнем изоляции, хотя теоретически многоверсионная архитектура могла бы обеспечить такой уровень изоляции.

Давайте перейдем к реально существующим в InterBase уровням изоляции. Сначала рассмотрим уровень *Read Committed*, задающийся константой read_committed. Транзакция, запущенная с таким уровнем изоляции, может читать изменения, произведенные из параллельно выполняющихся транзакций. Этот уровень изоляции часто используется для получения самого "свежего" состояния базы данных.

Как видно из таблицы 1.4, существуют две разновидности этого уровня изоляции: read_committed rec_version и read_committed no_rec_version. По умолчанию используется вариант с параметром rec_version. Это означает, что при чтении какой-либо записи просто считывается последняя подтвержденная версия записи.

Вариант с no_rec_version более сложен для объяснения. Вообще говоря, суть использования уровня read_committed с опцией no_rec_version сводится к тому, что транзакция будет не только пытаться считать самую последнюю подтвержденную версию записи, но и требовать, чтобы не было более свежей неподтвержденной версии.

При чтении записи в такой транзакции производится проверка, не существует ли у этой записи неподтвержденной версии. Если существует, то происходит следующее (в зависимости от того, какой режим блокировки выбран):

- Если wait, то наша транзакция ждет, пока не завершится транзакция, в которой создана неподтвержденная запись. И если она подтвердилась или отменилась, то считывается последняя подтвержденная версия.
- Если блокировка nowait, то немедленно возникает ошибка "Deadlock".

Очевидно, что уровень изоляции read_committed no_rec_version может привести к множеству конфликтов, и использовать его следует с большой осторожностью.

Уровень изоляции *SNAPSHOT* задается параметром concurrency. Можно сказать, что SNAPSHOT – самый "родной" режим InterBase, при котором преимущества версионности проявляются наиболее полно. При его использовании транзакция делает "снимок" маски транзакций в базе данных на момент запуска, и поэтому, пока она длится, видит те же самые данные, которые существовали на момент ее запуска. Никакие изменения, которые делаются параллельно выполняющимися транзакциями, ей не видны. Ей видны только свои изменения. При попытке в этой транзакции изменить данные, измененные другими транзакциями уже после ее запуска (имеются в виду как уже подтвержденные, так и еще неподтвержденные данные), возникает конфликт.

Пока выполняется транзакция с уровнем изоляции concurrency, удерживаются все версии записей на момент ее запуска, так как конкурирующие транзакции видят, что SNAPSHOT активен и не имеют права собрать версии записей, так как они могут (гипотетически) понадобится нашему SNAPSHOT.

Обычно SNAPSHOT применяется либо для длительных по времени запросов (отчетов), либо для организации блокирования записей, чтобы предотвратить их одновременное редактирование/удаление другими транзакциями.

Уровень изоляции *SNAPSHOT TABLE STABILITY* задается параметром consistency. Этот уровень изоляции аналогичен уровню SNAPSHOT, но дополни-

тельно блокирует таблицу на запись. Суть идеи проста – если транзакция с уровнем изоляции consistency проводит изменения на какой-либо таблице, то транзакции с уровнями изоляции read_committed и concurrency могут только читать эту таблицу, а транзакции с таким же уровнем изоляции (т. е. consistency) не смогут даже читать.

Очевидно, что использование этого уровня изоляции позволяет организовать последовательные (сериализуемые) обновления таблицы. Обычно такой уровень изоляции используется только для коротких обновляющих транзакций. Транзакция запускается, проводит очень короткое по времени изменение и сразу завершается. Другие транзакции в зависимости от режима блокировки wait или nowait либо ждут своей очереди, либо возбуждают исключение.

Рекомендации по использованию параметров транзакций

Как использовать транзакции – с этим вопросом часто сталкиваются начинающие разработчики. Конечно, для каждой конкретной задачи нужно решать вопрос индивидуально. Обычно все запросы к базе данных подразделяются на группы – запросы на чтение самого "свежего" состояния базы данных, запросы на текущие изменения, запросы на чтение справочных таблиц, запросы на чтение данных для построения отчета и т. д. Для каждой группы запросов обычно устанавливается своя транзакция (или группа транзакций) с набором параметров, нужных для выполнения задачи.

Рассмотрим типичное приложение базы данных, с помощью которого пользователь желает читать и изменять данные. В приложении имеется сетка (dbGrid в Delphi/C++Builder), в которой пользователь просматривает текущее содержание какой-то таблицы. Сетка содержит lookup-поля, которые заполняются значениями из справочников. Когда пользователь находит запись, которую нужно изменить (или просто желает добавить запись в таблицу), то он нажимает кнопку добавления/редактирования и в появившемся диалоге заполняет/изменяет поля записи и затем сохраняет/отменяет редактирование.

Как же настроить транзакции для такого приложения?

Для запроса SELECT…, который читает данные в сетку, следует использовать транзакцию с доступом "только для чтения" с уровнем изоляции READ COMMITED, чтобы получить самые "свежие" данные из таблицы, как только они будут обновлены/добавлены (не надо забывать о том, что наше приложение многопользовательское и одновременно могут работать несколько приложений). Примерный набор параметров такой:

```
read
read_committed
rec_version
nowait
```

При этом обеспечивается чтение всех подтвержденных другими транзакциями записей, причем без конфликтов с параллельно работающими пишущими и читающими транзакциями.

Такую транзакцию можно длительное время держать открытой – сервер не нагружается версиями записей.

Для запроса на изменение/добавление данных можно использовать транзакцию с уровнем изоляции concurrency. Запрос на обновление в этом случае должен быть очень коротким: пользователь заполняет необходимые поля, запускается транзакция, делается попытка выполнить запрос, и затем, если не возникло конфликта на запись с другой транзакцией, подтверждение нашей транзакции или откат, если был конфликт (на уровне клиентского приложения конфликты проявляются в виде исключений, которые удобно отлавливать с помощью конструкций try…except или try…catch).

Параметры такой транзакции будут следующими:

```
write
concurrency
nowait
```

Такой набор параметров позволит нам сразу (nowait) выявить то, что запись редактируется/изменяется другим пользователем (возникнет ошибка), а также предотвратить попытки других пользователей начать изменение записей, трансформированных нашей транзакцией (у претендента возникнет ошибка "update conflict"). Надо отметить, что перед редактированием нужно перечитать запись, потому что она могла быть изменена, а в кеше сетки может все еще находиться старая версия.

Для запросов, которые применяются для построения отчетов, однозначно нужно использовать транзакцию с режимом доступа "только для чтения" и с уровнем изоляции concurrency:

```
read
concurrency
nowait
```

Такая транзакция будет возвращать строго те данные, что существовали на момент ее запуска, – это очень важная особенность для отчетов, которые строятся за несколько проходов по базе данных.

Для запросов на чтение справочных данных можно использовать транзакцию, аналогичную запросу SELECT для выборки данных в сетку.

За пределами транзакций

Мы рассмотрели общие вопросы, связанные с транзакциями, а также особенности их практического применения в базе данных. В самом начале главы было сказано, что все действия в InterBase выполняются в контексте транзакций.

Однако существуют объекты, про которые говорят, что они находятся вне контекста транзакций. Это генераторы и внешние таблицы.

Генератор, как было описано в главе "Таблицы. Первичные ключи и генераторы", является счетчиком, хранящим некоторое целочисленное значение. Однако по своей реализации генератор является объектом совершенно уникальным. В отличие от остальных данных в базе данных значения генераторов хранятся на самом низком физическом уровне – на особых страницах генераторов. Это позволяет одновременно всем транзакциям одновременно видеть значения генераторов в любой момент времени. Это очень ценная возможность, которая

позволяет организовать бесконфликтные конкурентные вставки в параллельно выполняющихся транзакциях.

Внешние таблицы представляют собой файлы, находящиеся за пределами основного файла базы данных. Над внешними таблицами позволены только операции вставки и выборки (INSERT/SELECT). Отсутствие обновлений во внешних таблицах позволяет отказаться от хранения версий записей в этих таблицах, поэтому там всегда находятся актуальные данные, что позволяет отказаться от применения механизма транзакций для работы с данными в этих таблицах.

Двухфазное подтверждение транзакций

В завершение этой главы хочется рассказать о механизме двухфазного подтверждения транзакций. Дело в том, что InterBase предлагает уникальную возможность организовывать распределенные транзакции между разными базами данных и даже разными серверами (пожалуйста, не путайте двухфазное подтверждение транзакций с гетерогенными запросами, которые невозможно выполнять в InterBase).

Суть двухфазного подтверждения состоит в том, что в клиентском приложении можно запустить транзакцию сразу на двух серверах. Фактически проще всего это сделать, привязав один компонент транзакции сразу к двум компонентам базы данных.

Такая транзакция запустится сразу на двух серверах. Чтобы синхронизировать процесс завершения этой транзакции, вводится особое состояние, называемое Prepared. Это состояние означает, что транзакция завершилась на одном сервере и готова перейти в состояние Committed, как только транзакции на остальных серверах также перейдут в состояние Prepared. Если же транзакция хотя бы на одном из участвующих в процессе серверов завершится Rollback, то все транзакции из состояния Prepared тоже откатятся.

Теперь ясно, отчего могут возникнуть лимбо-транзакции, о которых упоминалось выше. Если между серверами разорвется соединение в тот момент, когда одна транзакция перешла в состояние Prepared и готова подтвердиться, то сервер не сможет решить, подтвердить или удалить изменения, сделанные этой транзакцией.

Не следует использовать двухфазное подтверждение транзакций на серверах, соединенных медленными каналами связи (модемами, например).

Заключение

Транзакции – один из наиболее сложных для понимания и объяснения вопросов в разработке приложений баз данных, независимо от того, о каком сервере баз данных идет речь. Поэтому изучение их применения является необходимой задачей для каждого разработчика приложений баз данных, если он хочет достигнуть вершин мастерства в своей профессии. Материал данной главы дает необходимые минимальные сведения о транзакциях и пищу для размышлений, однако для полного понимания транзакций следует обратиться к специальным статьям, посвященным различным аспектам этого вопроса. Эти статьи всегда можно найти на сайтах www.InterBase-world.com и www.ibase.ru.

Обзор библиотек доступа к InterBase

В данной главе мы рассмотрим существующие библиотеки доступа к InterBase и коротко охарактеризуем их свойства. Под "библиотекой доступа" будем понимать набор средств разработки, позволяющий разработчикам приложений баз данных InterBase создавать свои программы.

Основа библиотек доступа к InterBase

Какова бы ни была библиотека доступа, для какой бы среды разработки она ни предназначалась, в любом случае основой является *InterBase API*. InterBase API предоставляет базовый набор функций низкого уровня для работы с базами данных. Таким образом, любая библиотека доступа представляет собой "обертку" (wrapper) над функциями API. Библиотеки доступа организовывают функции API в соответствии с идеологией среды разработки, для которой предназначена библиотека.

Тем не менее нужно сказать, что, имея общую основу, все библиотеки доступа зачастую принципиально отличаются друг от друга. Любой опытный программист, попробовавший разрабатывать приложения с использованием различных библиотек, сможет рассказать о множестве отличий.

Библиотеки доступа

В мире приложений баз данных сложилось несколько устоявшихся подходов к работе с базами данных. Для InterBase можно выделить около пяти таких подходов: это работа с базой данных через BDE (Borland Database Engine, см. глоссарий), использование ODBC, применение OLE DB (ADO), работа через dbExpress и библиотеки прямого доступа.

Долгое время использование BDE для доступа к базам данных InterBase было фактически монопольной технологией. BDE представляет обобщенный механизм работы сразу со многими SQL-серверами, в том числе и с InterBase. Технология BDE поддерживалась в основном в продуктах компании Borland: Delphi, C++ Builder и др. Преимущество BDE, состоящее в универсальности подхода к работе с различными SQL-серверами, что значительно облегчает перенос приложений с одного сервера на другой, является также и его недостатком. Прежде всего это невозможность воспользоваться уникальными особенностями каждого SQL-сервера. Реализованная в BDE модель управления транзакциями, основной целью которой было облегчить перенос приложений с Paradox, не отвечала требованиям современных клиент-серверных приложений. Таким образом, в целом BDE не может рассматриваться как эффективная библиотека доступа именно к InterBase, хотя, как уже было сказано, в определенных условиях данный механизм может оказаться удобным. Тем не менее в этой книге мы не будем рассматривать работу с InterBase через BDE, тем более что этот вопрос освещается практически в любой книге о разработке приложений баз данных при помощи Borland Delphi.

ODBC (Open Database Connectivity) является одним из наиболее распространенных стандартов, которые обеспечивают доступ к базам данных. В отличие

от BDE, ODBC позволяет разрабатывать приложения баз данных InterBase практически в любых средах программирования, а не только в продуктах компании Borland. В настоящее время существует несколько ODBC-драйверов, поддерживающих все функции семейства InterBase 6.x и эффективно реализующих работу с базами данных. Работа с одним из наиболее распространенных ODBC-драйверов – Gemini – коротко описана в данной книге.

Несмотря на существование ODBC, корпорация Microsoft в настоящее время продвигает новый ключевой механизм доступа – технологию OLE DB. Разумеется, существуют продукты, поддерживающие эту технологию для InterBase. В данной книге мы рассмотрим, в частности, IBProvider – мощную OLE DB-библиотеку для работы с InterBase. Разработчики приложений на Visual C++, Visual Basic, ASP и других популярных систем могут создавать с помощью IBProvider эффективные приложения, использующие все преимущества как технологии OLE DB, так и СУБД InterBase. В числе уникальных возможностей IBProvider хочется отметить возможность прозрачной интеграции данных баз данных InterBase с базами данных MS SQL Server.

Технология dbExpress, появившаяся в последних версиях продуктов компании Borland (Delphi, C++Builder и Kylix), позволяет проектировать приложения, переносимые между различными SQL-серверами, и в то же время в полной мере использовать уникальные особенности каждого из серверов. Тем не менее на сегодняшний день существует не так много приложений, использующих эту новую технологию, поскольку разработчики все еще предпочитают устоявшиеся методы работы с базами данных InterBase. Мы не станем рассматривать использование dbExpress, хотя возможно, что этот вопрос будет включен в следующие издания этой книги.

И наконец, мы можем сказать несколько слов о библиотеках прямого доступа – наиболее эффективном способе работы с базами данных InterBase. Используя такие библиотеки, можно добиться минимального количества "прослоек" между непосредственно кодом клиентского приложения и вызовами InterBase API. Это позволяет получить в приложениях максимальную производительность; полную поддержку всех возможностей InterBase, а также минимальный объем конечного продукта – для работы приложений на базе библиотек прямого доступа требуется лишь минимальная установка клиента InterBase (см. главу "Установка InterBase – взгляд изнутри" (ч. 4)).

В нашей книге подробно рассмотрено применение наиболее прогрессивной, по мнению авторов, библиотеки доступа – Devrace™ FIBPlus™, которая поддерживает все версии InterBase, начиная с 4.x, а также все клоны InterBase: Firebird и Yaffil.

Несомненно, нельзя забывать про использование баз данных в приложениях, ориентированных на Интернет. Прежде всего это работа с базами данных в Java, а также в CGI-приложениях. Этим вопросам в книге посвящены две главы, рассматривающие простые примеры работы с базами данных InterBase.

Для популярных языков Perl, Python и PHP также существуют свои собственные библиотеки доступа к базам данных InterBase, с помощью которых можно легко построить динамический сайт с поддержкой базы данных. Эти библиотеки перечислены ниже в таблице 1.6, содержащей общий список всех библиотек доступа к InterBase.

Завершая этот обзор, необходимо добавить, что большинство библиотек доступа поддерживают возможность разработки кросс-платформенных клиентских приложений, которые можно легко переносить с Windows на Linux или другие ОС.

Список библиотек доступа к InterBase

Широкое распространение InterBase и его клонов по всему миру и использование в самых различных ипостасях привело к тому, что было создано множество библиотек доступа к InterBase/Firebird, ориентированных на самые различные среды программирования. Ниже, в таблице 1.6, приведен список наиболее популярных продуктов:

Таблица 1.6

Библиотека	Краткое описание	Где взять
Free IB Components (FIBC)	Четыре компонента, написанные в 1998 г. Грегори Детцем. Идеи, заложенные в FIBC, послужили основой для создания библиотек FIBPlus и IBX	ftp.ibphoenix.com /download
FIBPlus	Библиотека прямого доступа, оформленная в виде компонентов, применяется в Delphi 3–6, C++ Builder 3–6, Kylix. Поддерживает интеграцию со стандартными data-aware-компонентами. Основана на коде Free IB Components	www.fibplus.net
IBObjects	Набор компонентов для прямого доступа, включающий также визуальные компоненты для работы с базой данных	www.ibobjects.com
InterBase Express (IBX)	Набор компонентов для работы с базами данных InterBase, позволяющий использовать data-aware-компоненты для представления данных. Продукт основан на коде FreeIBComponents и входит в стандартную поставку Borland Delphi/C++ Builder Enterprise Edition	codecentral.borland.com/ codecentral/ccweb.exe/ author?authorid=102

Библиотека	Краткое описание	Где взять
Zeos Database Objects	Набор компонентов для работы с различными серверами баз данных, в том числе и InterBase. Позволяет использовать стандартные data-aware-компоненты для представления данных, а также содержит свои собственные визуальные компоненты	www.zeoslib.org
SQLAPI++	Библиотека классов для C++, позволяющая работать со многими SQL-серверами, в том числе и InterBase	www.sqlapi.com
Open Source Firebird and InterBase ODBC Driver	ODBC-драйвер для InterBase/Firebird. Существует в виде открытых кодов. Также поддерживает возможность организации "моста" ODBC-JDBC	www.ibphoenix.com /ibp_60_odbc.html
Gemini InterBase ODBC Driver	ODBC-драйвер, поддерживает все версии InterBase, начиная с 4.x, а также предоставляет поддержку всех возможностей InterBase 6.5 и Firebird 1.0	www.ibdatabase.com
IBProvider	OLE DB-провайдер для доступа к базам данных InterBase. Полностью поддерживает все свойства InterBase 6.x/Firebird 1.0	www. Ibprovider.com
SIBProvider	OLE DB-провайдер для доступа к базам данных InterBase	www.sibprovider.com
IBPerl	Объектно-ориентированная библиотека для Perl5 для работы с базами данных InterBase	www.karwin.com/ibperl
DBD::InterBase	DBI-драйвер для InterBase	dbi-InterBase.sourceforge.net

Библиотека	Краткое описание	Где взять
PHPLib for InterBase	Библиотека доступа к InterBase для языка PHP	www.intelicom.si
ADODB - InterBase PHP4 Data-base Wrapper	Библиотека доступа к InterBase для языка PHP	php.weblogs.com/adodb
Zope Driver for InterBase	Библиотека доступа к InterBase для языка Python	www.zope.org/Members /mcdonc/RS/InterBaseStorage
InterClient	JDBC-драйвер для досту-па к базам данных InterBase из Java	Включен в стандартную по-ставку Borland InterBase

Самый полный и "свежий" список библиотек доступа всегда можно найти на сайте поддержки данной книги www.InterBase-world.com, а также на сайте www.ibase.ru

Часть 2
Разработка приложений баз данных InterBase на Borland Delphi/C++ Builder/Kylix

Что такое InterBase Express?

Пользователи Borland Delphi 5–7 и Borland C++ Builder 5–6 редакций Professional и Enterprise наверняка уже обратили внимание на закладку InterBase в палитре компонентов. Именно эта закладка и представляет собой набор компонент под общим названием InterBase Express или IBX. Это компоненты для работы с базами данных InterBase, которые используют прямое InterBase API, т. е. обращаются к серверу непосредственно, без каких-то промежуточных (middle-ware) средств.

Фактически если разрабатывать приложения баз данных с использованием IBX, то для работы таких приложений нужно лишь наличие GDS32.DLL на диске в доступном месте.

Официально разработка IBX ведется в рамках IPL – InterBase Public License, т. е. компоненты доступны в полных исходных текстах и вы можете использовать их совершенно бесплатно. Однако фактически данные компоненты разрабатываются только сотрудниками корпорации Borland, совместимы только с определенными версиями и редакциями Delphi и C++ Builder и недоступны для публичной разработки, как это принято в обычных Open Source-проектах.

Компоненты IBX позволяют разрабатывать приложения, ориентированные на идеологию и архитектуру InterBase. К особенностям IBX можно отнести:

- явное управление транзакциями;
- поддержке расширений InterBase 6.0–7.0;
- поддержку event-alerters;
- использование генераторов для значений ключевых полей;
- управление сервером через Services API;
- поддержку стандартных и сторонних визуальных компонентов отображения данных;
- поддержку встроенных и сторонних генераторов отчета;
- совместимость с Delphi 5–7, C++ Builder 5–6 и Kylix

Основой кода IBX является библиотека FreeIBComponents, написанная Грегори Дилтцом в 1998 году. Основные изменения, сделанные в Borland, касались поддержки нового стандарта идентификаторов в SQLDialect 3, а также приведение компонентов к виду, аналогичному существовавшим компонентам (мы поясним суть этой аналогии позднее). Теперь с выходом каждой очередной версии Delphi или C++ Builder

IBX включается в поставку, однако все равно желательно проверять наличие исправлений или дополнительных сборок на сайте http://codecentral.borland.com.

Общее описание основных компонентов, включенных в состав IBX

 TIBDatabase – предназначен для подключения к базе данных. Основные методы: Open, Close.

 TIBTransaction – предназначен для явного управления транзакцией. Основные методы: StartTransaction, Commit, Rollback, CommitRetaining, RollbackRetaining.

 TIBTable – аналог стандартного TTable. Компонент предназначен для получения данных из одной таблицы или представления базы данных. Основное свойство – TableName. Основные методы: Open, Close. Набор данных, полученных при помощи TIBTable, является редактируемым, если речь идет о таблице базы данных или обновляемом представлении. Компонент совместим с визуальными компонентами.

 TIBQuery – аналог стандартного TQuery. Компонент предназначен для получения данных на основе SQL-запроса. Этот набор данных не всегда будет редактируемым, зачастую необходимо использовать дополнительный компонент TIBUpdateSQL, чтобы иметь возможность редактировать полученные сведения. Основное свойство – SQL. Основные методы: Open, Close, ExecSQL. Компонент совместим с визуальными компонентами.

 TIBDataSet – предназначен для получения и редактирования данных, является потомком стандартного класса TDataSet и полностью совместим со всеми визуальными компонентами. Основные методы: Prepare, Open, Close, Insert, Append, Edit, Delete, Refresh.

 TIBStoredProc – предназначен для выполнения хранимых процедур и получения набора данных на основе результатов выполнения процедуры. Получаемый набор данных является нередактируемым. Компонент совместим с визуальными компонентами. Основное свойство – StoredProcName. Основной метод – ExecProc.

 TIBUpdateSQL – аналог TUpdateSQL. Используется в паре с TIBQuery и предназначен для создания модифицируемых наборов данных. Основные свойства: DeleteSQL, InsertSQL, ModifySQL и RefreshSQL.

 TIBSQL – предназначен для выполнения SQL-запросов. В отличие от TIBQuery или TIBDataSet, TIBSQL не имеет локального буфера для набора данных и несовместим с визуальными компонентами.

 TIBDatabaseInfo – позволяет получить системную информацию о некоторых свойствах базы данных, соединения и сервера. Например, UserNames – список пользователей, подключенных к базе данных, PageSize – размер страницы базы данных.

 TIBSQLMonitor – предназначен для перехвата и отслеживания всех запросов, которые выполняют приложения, использующие IBX.

 TIBEvents – предназначен для получения пользовательских событий InterBase. Основное свойство – Events. Основные методы: RegisterEvents, UnregisterEvents.

Компоненты-оболочки для Services API

 TIBConfigService – предназначен для настройки параметров базы данных.

 TIBBackupService – предназначен для создания резервных копий (backup) баз данных.

 TIBRestoreService – предназначен для восстановления базы данных из резервной копии.

 TIBValidationService – предназначен для проверки целостности базы данных и согласования внутренних данных о транзакциях.

 TIBStatisticalService – предназначен для получения статистики о базе данных.

 TIBLogService – предназначен для создания и просмотра лог-файла работы сервера.

 TIBSecurityService – предназначен для редактирования списка пользователей на сервере.

 TIBLicensingService – предназначен для добавления и удаления сертификатов, регулирующих количество и свойства клиентских подключений к серверу InterBase.

 TIBServerProperties – предназначен для получения информации о сервере, параметров конфигурации и т. д.

 TIBInstall – предназначен для установки InterBase installation-компонента.

 TIBUnInstall – предназначен для установки InterBase un-installation компонента.

Использование основных компонентов InterBase eXpress (IBX)

Исторически сложилось так, что первое издание книги не содержало материалов по IBX. То есть данная глава написана специально для второго издания. После выхода книги мы получили ряд отзывов, которые наглядно показали нам, что многие разработчики (особенно те, кто впервые работает с IBX или FIBPlus) не представляют, как в целом взаимодействуют компоненты IBX между собой. В итоге, несмотря на аккуратное воспроизведение всех примеров из главы по FIBPlus, некоторые программисты не могут ни на шаг отойти от описанных ситуаций, поскольку спотыкаются буквально на совершенно очевидных вопросах. Чтобы осветить технологию несколько с других позиций, мы решили спланировать материал этой главы немного иначе, чем это было сделано с материалами по FIBPlus.

Иерархия компонентов в IBX

Поскольку вы работаете с Delphi (или с C++ Builder), то предполагается, что вы знакомы с объектно-ориентированным программированием. Таким образом, разобравшись, как именно и от кого унаследованы различные компоненты IBX, можно будет более полно представить себе, как именно их нужно использовать. Рассмотрим рис. 2.1.

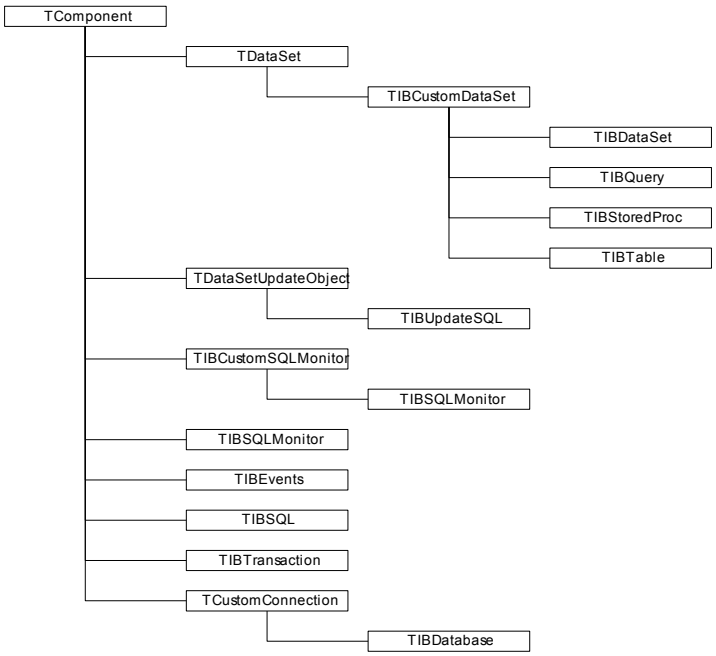


Рис. 2.1. Иерархия компонентов IBX

Внимательно рассмотрев эту схему, можно сделать сразу несколько выводов.

Во-первых, очевидна несовместимость IBX с версиями Delphi меньше 5, поскольку класс TCustomConnection появился лишь в Delphi 5.

Во-вторых, становится ясно, почему компонент TIBSQL невозможно использовать вместе с визуальными db-aware-компонентами вроде TDBGrid или TDBEdit. Все стандартные визуальные db-aware-компоненты работают только с потомками класса TDataSet. Поэтому для db-aware компонентов невозможна связка с TIBSQL, который не унаследован от TDataSet.

Из той же схемы видно, что в IBX есть компоненты, совместимые с db-aware-компонентами (TDBGrid и т. д.). Это потомки внутреннего класса TIBCustomDataSet – TIBDataSet, TIBTable, TIBQuery и TIBStoredProc. Вообще говоря, почти вся данная "ветка" классов по своему назначению близка к аналогичной ветке компонентов для работы с BDE – TTable, TQuery и TstoredProc – и предназначена для "быстрой" миграции старых приложений с BDE на IBX.

> Также следует обратить внимание на компонент TIBUpdateSQL, который является аналогом компонента TUpdateSQL, предназначенного для работы с BDE.

Судя по отзывам пользователей, переходящих с BDE на IBX, в действительности такое сопоставление компонентов IBX и BDE не всегда приносит желаемый результат, так как взаимозаменяемость старых BDE-компонентов на новые аналоги из IBX зачастую противоречит рекомендациям специалистов в силу различий в идеологии.

Прежде всего это связано с управлением транзакциями и обработкой большого количества записей. Ниже мы остановимся на этом вопросе подробнее, а теперь перейдем к рассмотрению особенностей компонентов TIBTable, TIBQuery и TIBStoredProc.

Особенности TIBTable, TIBQuery и TIBStoredProc

Фактически, компонент TIBCustomDataSet имеет всю необходимую функциональность для получения базы данных InterBase и поддерживает возможность редактирования этой информациии с помощью визуальных db-aware-компонентов.

Для выборки данных, их изменения, удаления и вставки в TIBCustomDataSet используется набор свойств, представляющих собой SQL-запросы для манипулирования данными, – это SelectSQL, DeleteSQL, InsertSQL и ModifySQL.

> Отдельно следует сказать о RefreshSQL. Этот запрос не используется для модификации записи, но является очень полезным для получения значений полей, которые были изменены триггерами базы данных и конкурирующими транзакциями.

В свойстве SelectSQL указывается запрос на выборку данных (SELECT… FROM…), которые будут доступны для просмотра и, в зависимости от содержимого остальных запросов, для редактирования, удаления и т. д.

В свойствах DeleteSQL, InsertSQL и ModifySQL указываются соответствующие запросы, которые будут вызываться автоматически самим компонентом при вызове операций Delete, Insert и Edit для удаления, вставки и редактирования записей.

Фактически все, что нужно сделать программисту, – это написать нужные запросы, выполняющие нужные операции над записями. Далее мы более подробно рассмотрим потомков TIBCustomDataSet.

TIBTable

Компонент TIBTable прячет все указанные выше свойства, а вместо этого пользователю предоставляется свойство TableName. Пользователь указывает имя таблицы в свойстве TableName, а компонент автоматически формирует набор "спрятанных" запросов.

Например, для таблицы с именем Table1 запрос в SelectSQL будет иметь вид:

```
SELECT * FROM Table1
```

Легко представить, что в нашей таблице несколько миллионов записей и этот запрос попытается получить их их в полном объеме на клиента. Например, при вызове Locate, который так любят пользователи BDE, если запись, соответствующая условиям поиска, не найдена в загруженном наборе записей, то TIBTable будет запрашивать оставшиеся записи, пока не найдется подходящая запись или пока не закончатся записи в таблице.

Очевидно, что это вызовет колоссальную нагрузку на SQL-сервер и клиента, особенно в многопользовательской среде. Ни один специалист не рекомендует использование компонента TIBTable в реальных программных проектах, предназначенных для управления серьезными базами данных в многопользовательской среде.

TIBQuery

Аналогично TIBTable-компонент TIBQuery скрывает запросы для получения и редактирования данных. Вместо скрытого в этом компоненте свойства SelectSQL разработчику предлагается использовать свойство SQL. На самом деле после присвоения свойства SQL компонент присваивает его значение свойству SelectSQL.

Но самое примечательное с точки зрения проектирования классов начинается тогда, когда мы хотим сделать наш запрос редактируемым (live-query).

Поскольку свойства DeleteSQL, InsertSQL и ModifySQL спрятаны, то TIBQuery сам по себе не может предоставить разработчику редактируемые запросы.

Однако, как уже было сказано, TIBQuery был сделан как аналог TQuery и для полной аналогии в IBX введен компонент TIBUpdateSQL. Он содержит собственные свойства DeleteSQL, InsertSQL и ModifySQL и может подключаться к TIBQuery. После чего TIBQuery начинает использовать свойства компонента TIBUpdateSQL для редактирования собственных данных! Получается, что готовую функциональность TIBCustomDataSet, уже заложенную в него с самого начала, приходится дублировать в отдельном компоненте.

> Отметим, тем не менее, что в данной связке имеется несомненный смысл, если речь идет о миграции готовых BDE-приложений на IBX. В общем-то, вероятно, только ради этого данные классы и были введены.

Мы можем принять это как выгоду с точки зрения замены старых BDE-компонентов на нечто похожее из IBX, однако, если вы пишете новое приложение, которое с самого начала базируется на IBX, мы бы рекомендовали вам использовать TIBDataSet как вместо TIBTable, так и вместо TIBQuery.

Некоторые разработчики используют TIBQuery для выполнения только модифицирующих запросов, т. е. тех, которые не возвращают результата. Например, это может быть запрос с предложением INSERT или DELETE.

Для выполнения запросов, не возвращающих результирующий набор данных, в TIBQuery предусмотрен метод ExecSQL. Но фактически, как видно из исходных текстов компонента, вызов данного метода не отличается от вызова метода Open, который унаследован от TIBCustomDataSet.

Поэтому для выполнения запросов, которые не возвращают результатов (в том смысле, что не возвращают набор результирующих строк), не имеет смысла использовать компоненты, предназначенные для взаимодействия с визуальными db-aware-компонентами. Вместо этого лучше использовать компонент TIBSQL, который не буферизует получаемые данные и предназначен именно для простого и быстрого выполнения SQL-запросов, в частности не возвращающих набор строк.

TIBStoredProc

Данный компонент предназначен для выполнения исполняемых (executed) хранимых процедур. Он также является потомком TIBCustomDataSet и полностью совместим с визуальными компонентами.

Являясь прямым наследником TIBCustomDataSet, компонент TIBStoredProc прячет все основные свойства предка и добавляет такое свойство, как ProcedureName.

> Следует понимать, что в данном случае слово "прячет" относится больше к идеологии проектирования классов, нежели к технической стороне дела. Очевидно, что спрятать свойства предка в Object Pascal нельзя. TIBCustomDataSet уже имеет все необходимые свойства, которые уже **готовы** для публикации, их просто не вынесли в секцию published. И хотя мы согласимся, что в данном случае нельзя однозначно сказать, насколько правильным является данное проектирование классов, однако считать его красивым довольно трудно.

В результате указания названия процедуры (например, proc1) компонент сформирует SQL-запрос следующего вида:

```
EXECUTE PROCEDURE Proc1
```

В целом возможно, что проще вызвать хранимую процедуру с помощью соответствующего SQL-запроса, используя TIBDataSet, если мы хотим показать результат в db-aware-компонентах, или TIBSQL, если мы хотим вызывать хранимую процедуру вида executable (т. е. не возвращающую набор данных).

Единственным преимуществом использования TIBStoredProc по сравнению с TIBDataSet является тот факт, что TIBStoredProc самостоятельно формирует список параметров процедуры по ее имени, обращаясь к системным таблицам.

Подключение к базе данных

Очевидно, что прежде чем начать работать с базой данных, надо к ней подключиться. Специально для этого в состав IBX включен компонент TIBDatabase.

Для наших примеров в этой главе мы будем использовать базу данных Employee.gdb, которая поставляется вместе с InterBase.

Поместите TIBDatabase на форму и вызовите контекстное меню (рис. 2.2).

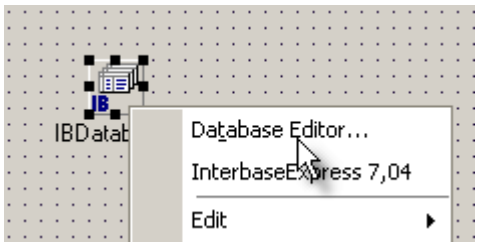


Рис. 2.2. Вызов редактора IBDatabase

Из рисунка видно, что у TIBDatabase существует специальный редактор, который позволяет вам настраивать определенные параметры компонента во время разработки приложения (рис. 2.3).

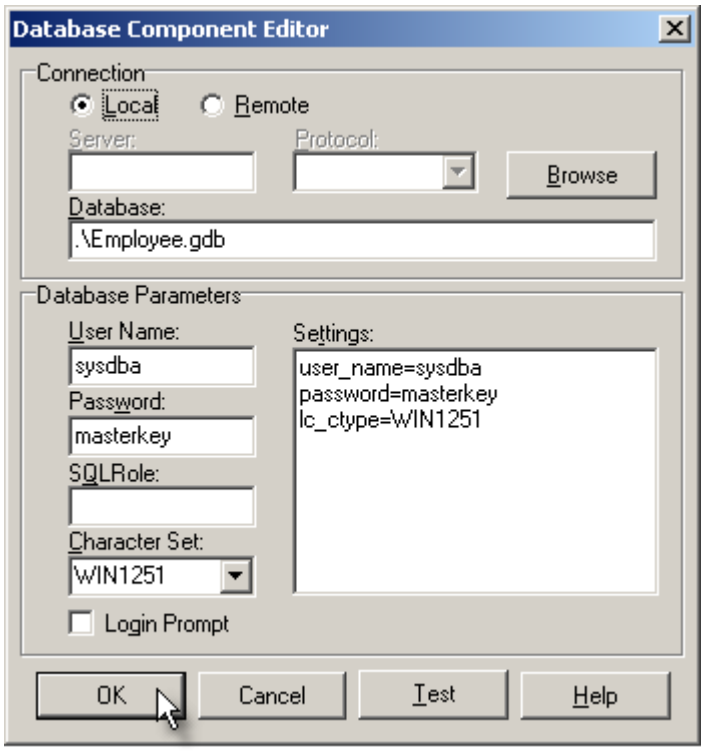


Рис. 2.3. Редактор TIBDatabase

В частности, необходимо указать путь к базе данных в разделе Connection. Путь включает в себя имя сервера и локальный путь к файлу базы данных на

сервере. В случае, если сервер локальный, то достаточно указать только путь к файлу базы данных.

> Немного подробнее относительно использования удаленного сервера. Предположим, что InterBase установлен у вас в сети на компьютере с сетевым именем SERV_IB и доступен по протоколу TCP/IP. Пусть база данных находится на сервере на диске E: в файле MY_DB.GDB. В данной ситуации в редакторе компонента вам надо будет указать тип подключения Remote, имя сервера SERV_IB, а путь к базе данных оставить локальным относительно сервера, т. е. , E:\ MY_DB.GDB. Это важный момент! Вы указываете серверу путь к локальному файлу базы на сервере, а не указываете вашей программе сетевой путь к базе данных. В сущности, именно об этом говорится в главе "Создаем базу данных" в разделе "Строка соединения", а здесь мы лишь хотим еще раз подчеркнуть, что данный редактор создает точно такую же строку соединения, которую вы вводите вручную, например, при использовании инструментов администратора и разработчика InterBase.

Вы можете протестировать параметры подключения не выходя из диалога при помощи кнопки Test. Если все указано правильно, то вы увидите сообщение Successful connection.

Управление транзакциями

Необходимо помнить, что любое действие с базой данных происходит в рамках той или иной транзакции. Работа с InterBase основана на явном управлении транзакциями, а поскольку библиотека IBX – это обертка вокруг соответствующих функций InterBase API, то использование этих компонентов также предполагает, что программист явным образом будет управлять транзакциями из своего приложения. Для контроля транзакций в IBX существует специальный компонент TIBTransaction (рис. 2.4):

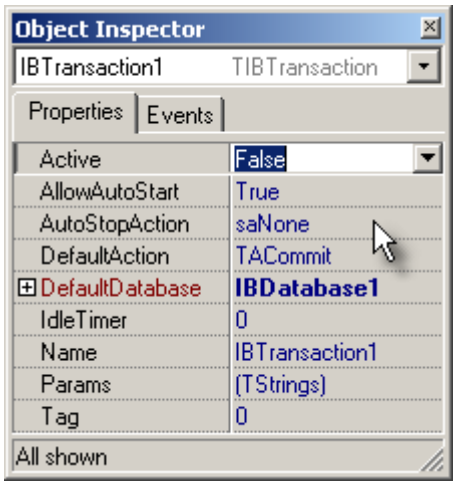


Рис. 2.4. Свойства TIBTransaction

Как видно из рисунка, компонент не слишком перегружен свойствами. Фактически основным является свойство Params, в котором можно указать уровень изоляции транзакции. Для этого необходимо знать соответствующие системные константы из InterBase API. Однако для большинства ситуаций вам вполне хва-

тит использования одного из четырех заранее заданных уровней изоляции. Для того чтобы выбрать один из них, можно вызвать редактор компонента (рис. 2.5).

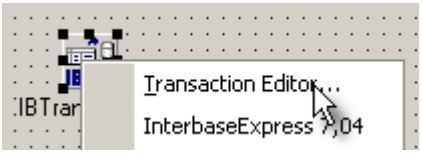


Рис. 2.5. Вызов редактора TIBTransaction

В появившемся диалоге вы сможете указать нужный уровень изоляции, а заодно и увидеть сразу, какими константами он задается (рис. 2.6).

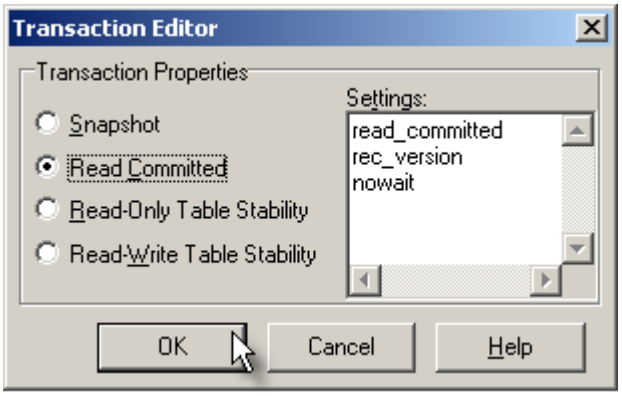


Рис. 2.6. Редактор TIBTransaction

Для большинства случаев рекомендуется использовать режим Read Committed, который позволит запросам в одной транзакции "видеть" изменения, сделанные и подтвержденные в контексте других транзакций.

TIBTransaction ссылается на компонент базы данных при помощи свойства DefaultDatabase. Если также указать свойство DefaultTransaction у TIBDatabase, то в дальнейшем любые компоненты (TIBDataSet, TIBSQL и т. д.), которые ссылаются на TIBDatabase, будут автоматически "подхватывать" и указанную транзакцию.

Теперь рассмотрим свойства AllowAutoStart и AutoStopAction. Как вам уже известно, любой запрос к базе данных должен выполняться в контексте транзакции. То есть, прежде чем выполнить даже простейший запрос вида SELECT * FROM TABLE1, необходимо предварительно запустить транзакцию при помощи вызова IBTransaction.StartTransaction.

Такой "ручной" вызов не всегда удобен. Более того, каждый раз совершенно определенно известно: если мы хотим выполнить запрос, то мы должны запустить транзакцию. Чтобы избежать лишнего кода, связанного с запуском транзакций, можно установить значение свойства AllowAutoStart равным True. В этом случае если мы попробуем, например, открыть TIBDataSet, то он сам автоматически запустит соответствующую транзакцию.

Аналогичный смысл имеет свойство AutoStopAction. Когда мы закрываем все запросы, выполнявшиеся в контексте автоматически запущенной транзакции, то

TIBTransaction выполняет действие, указанное в AutoStopAction. Например, автоматически подтверждает всю транзакцию при помощи метода Commit, если свойство AutoStopAction равно saCommit. Таким образом, разработчику предоставляется возможность указать, как компоненты должны автоматически взаимодействовать друг с другом!

Выполнение запросов при помощи TIBDataSet

> В данной главе сознательно не рассматривается работа с TIBTable или TIBQuery, так как авторы книги не считают их использование целесообразным. Если вы все же желаете использовать эти компоненты (например, для облегчения миграции приложения с BDE), то для изучения их функций можно обратиться к документации по аналогичным BDE-компонентам, а также к документации по IBX.

Итак, поместите на форму следующие компоненты:

```
IBDataSet1: TIBDataSet;
DataSource1: TdataSource
DBGrid1: TDBGrid
```

Необходимо указать свойства Database и Transaction у IBDataSet1, задать DataSource1.DataSet равным IBDataSet1, а также указать DBGrid1.DataSource равным DataSource1.

Теперь необходимо указать тот запрос, который мы хотим выполнить, в свойстве SelectSQL у IBDataSet1 (рис. 2.7).

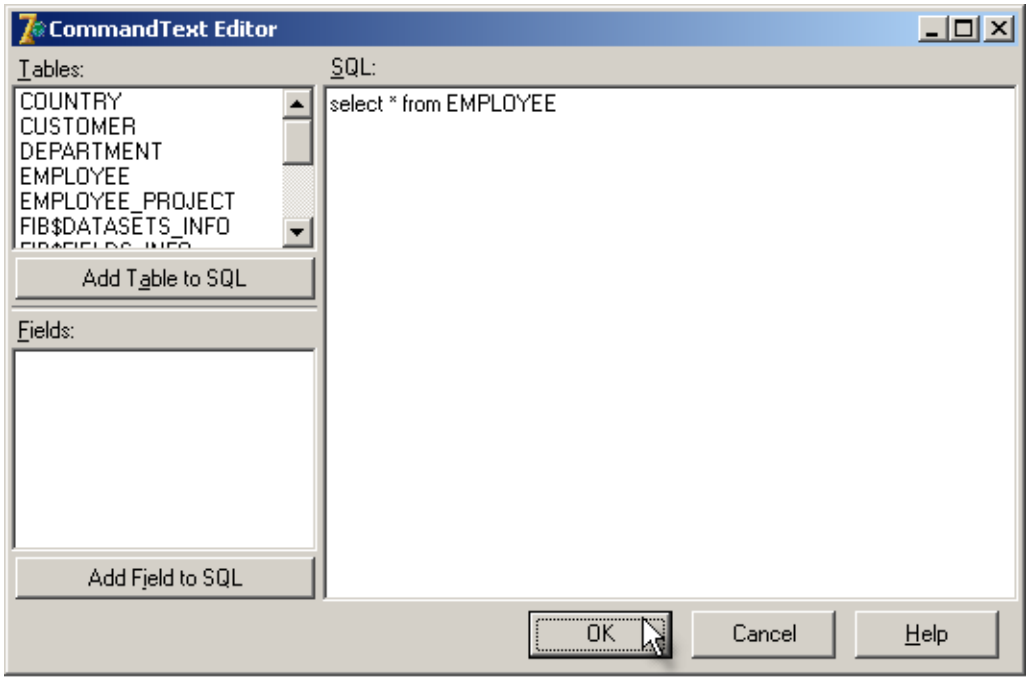


Рис. 2.7. Редактор свойства SelectSQL

Теперь мы можем открыть запрос прямо в design-time, задав свойство Active в True. Результат наших действий приведен на рис. 2.8.

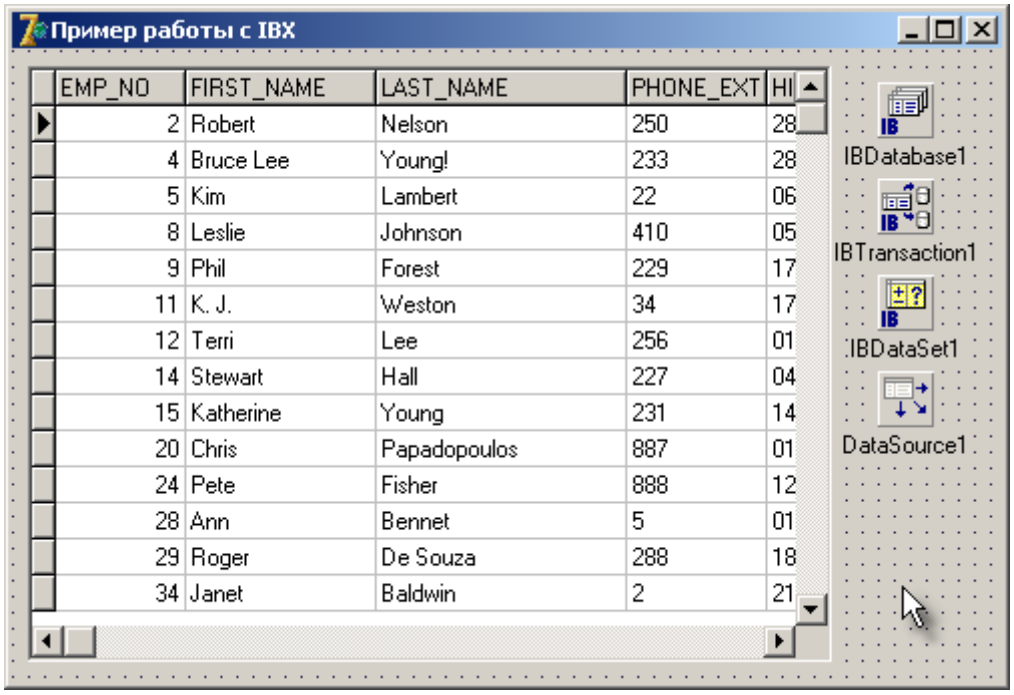


Рис. 2.8. Открытие запроса в IBDataSet1

Чтобы получить то же самое во время выполнения программы, напишите следующий обработчик события OnFormCreate у основной формы приложения:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  IBDatabase1.Open;
  IBDataSet1.Open;
end;
```

> Не забудьте закрыть подключение к базе данных перед сохранением проекта, иначе попытка IBDatabase1.Open вызовет ошибку. Вы также можете написать этот код иначе: IBDatabase1.Connected := True; IBDataSet1.Active := True. Он не вызовет сообщения об ошибке, если IBDatabase1 и IBDataSet1 были активны в design-time.

Редактируемые запросы

Редактирование данных при помощи визуальных компонентов

Если вы уже запустили пример, представленный выше, и попробовали исправить хотя бы одну запись в таблице, то наверняка получили сообщение, что сделать это невозможно. Причины этого сообщения очевидны, поскольку компонент IBDataSet1 имеет только свойство SelectSQL. Это свойство содержит за-

прос на выборку записей, его выполнение позволяет получить список тех записей из таблицы EMPLOYEE с сервера, которые удовлетворяют условию в WHERE.

А для того чтобы, к примеру, исправить какую-либо запись, необходимо выполнить команду UPDATE. В общем случае выполнение этой команды может вообще не зависеть от нашего запроса в SelectSQL.

Но поскольку TIBDataSet был спроектирован в первую очередь для того, чтобы использоваться совместно со стандартными визуальными компонентами, то он предоставляет нам средства для автоматического выполнения тех модифицирующих запросов, которые необходимы для изменения данных, полученных при помощи SelectSQL. Чтобы окончательно не запутаться, давайте подробнее рассмотрим этот вопрос.

Итак, мы открыли запрос в SelectSQL. Как видно из рисунка, первая полученная запись содержит имя сотрудника Robert. Предположим, мы хотим исправить это имя на John.

В момент изменения произойдет следующее: в локальном буфере IBDataSet1 у текущей записи значение поля FIRST_NAME будет изменено с Robert на John. В базе данных данное изменение пока никак не отражается. Чтобы произвести фактическое изменение данных на сервере, необходимо выполнить соответствующий запрос с UPDATE. Этот запрос необходимо заранее указать у компонента IBDataSet1 в свойстве ModifySQL:

```
UPDATE EMPLOYEE
SET
  EMP_NO = :EMP_NO,
  FIRST_NAME = :FIRST_NAME,
  LAST_NAME = :LAST_NAME,
  PHONE_EXT = :PHONE_EXT,
  HIRE_DATE = :HIRE_DATE,
  DEPT_NO = :DEPT_NO,
  JOB_CODE = :JOB_CODE,
  JOB_GRADE = :JOB_GRADE,
  JOB_COUNTRY = :JOB_COUNTRY,
  SALARY = :SALARY
WHERE
  EMP_NO = :OLD_EMP_NO
```

Как видно из примера, вместо реальных значений в этом запросе указаны параметры, названия которых совпадают с названием реальных полей. Таким образом, когда пользователь изменит значения полей конкретной записи, то IBDataSet1 сам задаст значения всех параметров, взяв их из соответствующих полей. В частности, значение параметра :FIRST_NAME будет равно John. Запрос из ModifySQL выполнится, и только после этого изменения, сделанные пользователем, окажутся в базе данных.

Аналогичная последовательность действий связана с запросами в свойствах InsertSQL и DeleteSQL – они выполняются при вставке новой записи и удалении записи пользователем.

Обратите внимание на префикс OLD_ в названии параметра :OLD_EMP_NO. Данный префикс означает, что IBDataSet должен подставить в параметр значение поля *до изменения пользователем*.

Итак, если мы сформируем все модифицирующие запросы, то наш IBDataSet позволит пользователям редактировать данные, т. е. мы фактически получим запрос, который разработчики на Delphi/C++Builder обычно называют "живым"(live query).

Существует еще одна важная особенность при создании "живых" запросов. После выполнения любого модифицирующего действия IBDataSet1 выполнит запрос, указанный в свойстве RefreshSQL. Этот запрос должен возвращать только одну запись – текущую и нужен для обновления значений полей текущей записи после сделанных исправлений:

```
SELECT
  EMP_NO,
  FIRST_NAME,
  LAST_NAME,
  PHONE_EXT,
  HIRE_DATE,
  DEPT_NO,
  JOB_CODE,
  JOB_GRADE,
  JOB_COUNTRY,
  SALARY,
  FULL_NAME
FROM EMPLOYEE
WHERE
  EMP_NO = :EMP_NO
```

Смысл данного запроса становится очевидным, если допустить существование в базе данных триггеров для таблицы EMPLOYEE, которые модифицируют значения полей. Поскольку изменения происходят в самой базе данных сразу после вставки или после изменения записей, то без повторного перечитывания записи (т. е. без Select только что вставленной или измененной записи), мы не узнаем о тех изменениях, которые были сделаны в триггерах. Можно, конечно, вообще переоткрыть весь запрос, заданный в SelectSQL.

Именно такой механизм и реализуется в BDE, когда мы вынуждены целиком переоткрывать все запросы. В IBX без этого легко можно обойтись, используя RefreshSQL, значительно сэкономив при этом сетевой трафик и снизив нагрузку на сервер, поскольку получение всего лишь одной измененной записи гораздо более эффективно, чем переоткрытие запроса целиком.

IBX предоставляет нам возможность быстро сгенерировать необходимые модифицирующие запросы при помощи редактора IBDataSet (рис. 2.9).

Выбрав из списка Table Name нашу таблицу и нажав кнопку Get Table Fields, мы сформируем списки Key Fields и Update Fields. В списке Key Fields нужно выделить те поля, которые будут формировать условие WHERE в наших запросах. Очевидно, что это должны быть поля, которые определяют первичный ключ у таблицы. Если такой ключ существует для выбранной таблицы, то вы можете просто нажать на кнопку Select Primary Keys, чтобы автоматически выделить нужные поля.

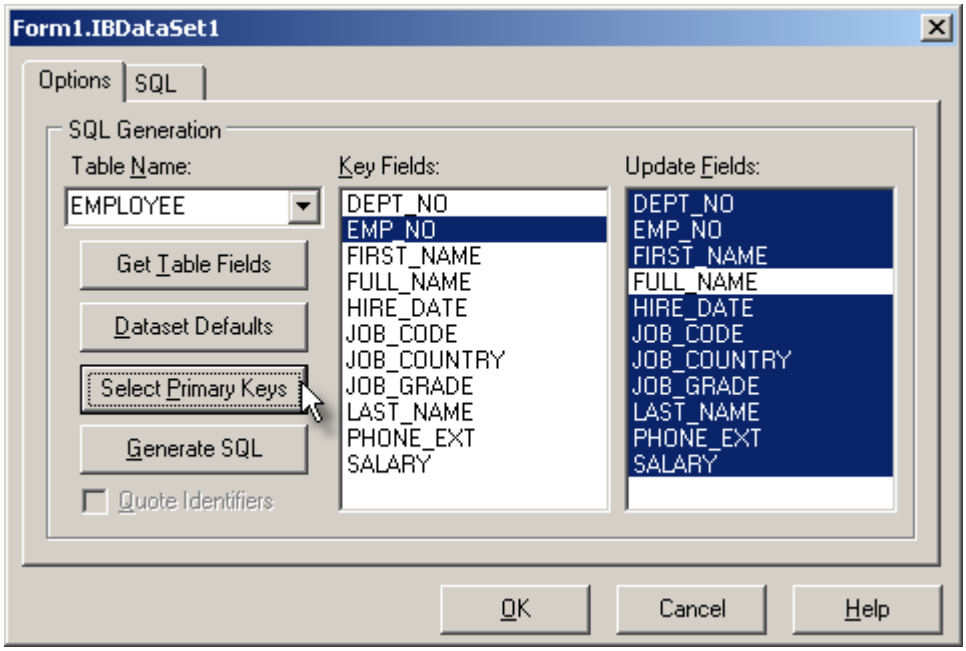


Рис. 2.9. Генератор модифицирующих запросов

В списке Update Fields необходимо выделить те поля, которые потом пользователь сможет редактировать. На рисунке видно, что поле FULL_NAME не включено в список, поскольку это CALCULATED-поле и его значение нельзя менять.

> К сожалению, надо самому точно знать, какие поля необходимо исключать из модифицирующих запросов, поскольку компоненты не дадут ни одной подсказки. Даже при подготовке данного раздела пришлось потратить впустую некоторое время, чтобы понять, почему IBDataSet1 никак не "хотел" становиться модифицируемым. Только выяснив, что поле FULL_NAME является вычислимым, стало понятно, почему возникает ошибка: IBDataSet1 пытался выполнить команду Prepare для каждого из запросов и сервер каждый раз сообщал, что не может изменить read-only-поле! Хочется отметить, что этой проблемы нет в генераторе запросов, реализованном в FIBPlus.

Остается нажать кнопку Generate SQL, чтобы получить все запросы: InsertSQL, ModifySQL, DeleteSQL и RefreshSQL.

Программное редактирование данных

Как показал некоторый опыт в поддержке пользователей, программисты, впервые применяющие IBX и аналогичные компоненты, не совсем понимают каким образом можно использовать TIBDataSet для того, чтобы управлять данными программно, а не просто давать пользователю возможность вносить какие-то исправления.

Например, периодически возникает вопрос: "А как мне автоматически вставить новую запись в IBDataSet? Может быть, стоит разместить рядом отдельный компонент IBSQL, в котором написать запрос INSERT, выполнить его, а потом переоткрыть IBDataSet? Или, может быть, можно синхронизировать новую за-

пись в IBSQL и существующие записи в IBDataSet? Существуют ли методы для прямого редактирования локального буфера записей в IBDataSet?"

Подобные "технологические" решения можно придумывать очень долго; поскольку фантазия разработчиков практически неистощима, мы можем найти выход из любой ситуации и обойти любое ограничение существующей технологии. Однако в данном случае решения "проблемы" проистекают от простого непонимания принципов работы IBDataSet. Поверьте, все гораздо проще.

В самом начале главы сказано, что TIBDataSet порожден классом TDataSet, а значит, наследует основные принципы его работы. У TIBDataSet как наследника TDataSet существует три основных метода для изменения данных: Delete, Insert (Append) и Edit.

Предположим, что мы хотим вставить новую запись в наш IBDataSet1 по нажатии кнопки:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with IBDataSet1 do begin
    Insert;
    FieldByName('EMP_NO').AsInteger := 147;
    FieldByName('DEPT_NO').AsInteger := 600;
    FieldByName('JOB_CODE').AsString := 'VP';
    FieldByName('JOB_GRADE').AsInteger := 2;
    FieldByName('SALARY').AsInteger := 105900;
    FieldByName('HIRE_DATE').AsDateTime := Now;
    FieldByName('JOB_COUNTRY').AsString := 'USA';
    FieldByName('FIRST_NAME').AsString := 'Иван';
    FieldByName('LAST_NAME').AsString := 'Иванов';
    Post;
  end;
end;
```

Теперь поясним, как это работает. После того как мы вызываем метод Insert, IBDataSet1 формирует пустой буфер для нашей (пока еще не введенной) записи. Далее мы задаем значения нужных полей при помощи вызовов метода FieldByName и заканчиваем (точнее, подтверждаем) редактирование вызовом метода Post. В этот момент IBDataSet1 выполняет запрос, прописанный в свойстве InsertSQL, подставив вместо параметров те значения полей, которые мы задали.

Если запрос выполнился успешно, то IBDataSet1 автоматически выполняет RefreshSQL для обновления только что вставленной записи – для проверки изменений, внесенных на стороне базы данных.

Аналогичным образом мы можем редактировать записи. Например, мы можем модифицировать все записи в нашем запросе:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with IBDataSet1 do begin
    First;
    while not Eof do begin
      Edit;
       FieldByName('LAST_NAME').AsString :=
     trim(FieldByName('LAST_NAME').AsString) + ', esquire';
```

```
      Post;
      Next;
    end;
  end;
end;
```

Принцип тот же самый, что и при вставке записей. Мы вызываем метод Edit, подготавливая буфер текущей записи для редактирования, потом меняем значения поля при помощи FieldByName, а потом заканчиваем редактирование, вызвав метод Post. В результате IBDataSet1 подставляет значения полей записи в ModifySQL и выполняет запрос.

> Оговоримся, что в реальной практике подобная обработка большого множества записей не является хорошим решением. Гораздо легче было бы выполнить одну команду UPDATE, которая добавила бы строку ', esquire' ко всем записям таблицы. Пример выше дан только для демонстрации метода Edit.

Как видно из примеров выше, нет никакой нужды ни в каких дополнительных компонентах, синхронизации о прочих непонятных "телодвижениях", если мы хотим редактировать данные в IBDataSet программно.

И снова про транзакции

Новичков иногда пугает "особенность" IBX закрывать все запросы при подтверждении или "откате" транзакции. Разместим на нашей форме две кнопки, как показано на рис. 2.10: Button1 (свойство Caption равно Commit) и Button2 (Rollback).

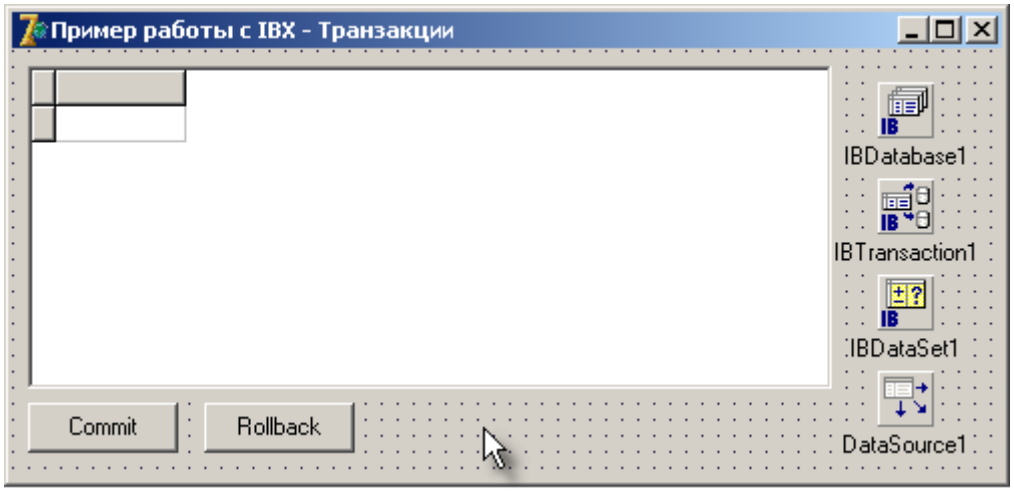


Рис. 2.10. Кнопки управления транзакцией

Далее напишем следующие обработчики событий нажатия на эти кнопки:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  IBTransaction1.Commit;
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  IBTransaction1.Rollback;
end;
```

Теперь если мы запустим приложение и нажмем любую из этих кнопок, то увидим, что после завершения транзакции наш запрос также будет закрыт.

Это действие совершенно очевидно, поскольку, как уже было сказано, любой запрос должен выполнятся только в рамках определенной транзакции и, таким образом, если транзакция уже закрыта, то запрос не может быть активным.

Однако для тех разработчиков, которые до сих пор пользовались BDE, такое поведение компонентов непривычно. Существует два метода. Первый: запоминать активные записи во всех открытых запросах перед закрытием транзакции, потом заново открывать все запросы и перемещать указатели текущих записей на "старое место". Именно так и работает механизм неявного переоткрытия, реализованный в BDE. Второй – это использовать методы CommitRetaining и RollbackRetaining, впервые появившиеся в InterBase 5.x. Эти методы в целом аналогичны методам Commit и Rollback, однако они реализованы таким образом, что сервер автоматически перезапускает транзакцию и не закрывает открытых запросов. Использование CommitRetaining и RollbackRetaining позволит вам избегать массовых операций переоткрытия ваших запросов.

Однако тут есть одна особенность, которая сразу не бросается в глаза. Предположим, что вы делали какие-то изменения, после чего решили отменить их и вызвали RollbackRetaining. Все модифицирующие запросы, которые были отправлены на сервер в контексте данной транзакции, будут отменены, однако локальный буфер TIBDataSet останется неизмененным.

Таким образом, содержимое, например, TDBGrid в вашем приложении будет отличаться от реального состояния таблицы на сервере. Поэтому мы рекомендуем вам все-таки заново открывать запросы, изменения которых были отменены при помощи RollbackRetaining.

Использование генераторов для автоинкрементных полей

Чаще всего автоинкрементные поля используются для поддержки суррогатных первичных ключей. В таблице заводится поле, значения для которого генерируются при помощи доступных технических средств, гарантирующих уникальность получаемых значений.

Существует несколько способов получения таких значений, мы же остановимся на том, который рекомендуется для большинства баз данных в InterBase. Как вы знаете, в InterBase существуют специальные объекты – генераторы, которые гарантируют получение уникальных возрастающих значений, независимых от контекста транзакций.

В IBDataSet существует специальное свойство, которое позволяет нам удобно использовать генераторы для создания уникальных значений первичного ключа, а именно GeneratorField. У этого свойства существует специальный редактор, доступный в design-time (рис. 2.11).

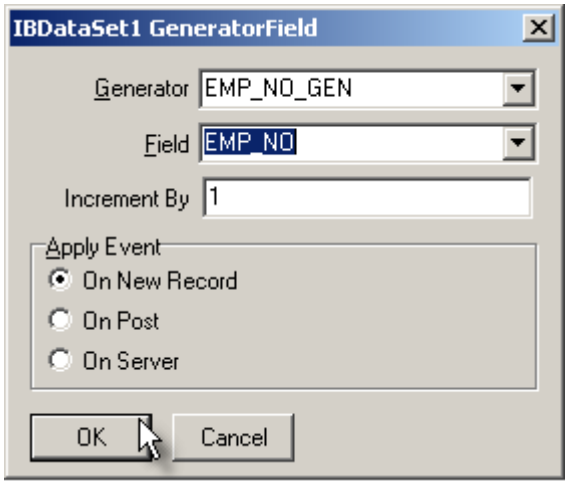


Рис. 2.11. Настройка автоинкрементного поля

Укажите название генератора, имя поля в таблице, значения для которого будут генерироваться, шаг увеличения счетчика (в большинстве случаев это 1), а также опцию, указывающую, когда будет генерироваться значение поля: сразу при вставке новой записи (On New Record), при завершении вставки (On Post) или при помощи триггера (On Server). В последнем случае вы не увидите значения сгенерированного поля, пока не переоткроете весь запрос, однако данная опция все равно может оказаться полезной. Поскольку наше поле EMP_NO входит в первичный ключ, то его значение не может формально быть незаданным (NULL). Если не указывать свойство GeneratorField, то IBDataSet1 будет требовать от нас указывать значение поля EMP_NO в обязательном порядке и мы не сможем "переложить ответственность" на триггер.

Указав же явным образом, что значение поля будет получено именно в триггере, мы обойдем это ограничение. Тем не менее мы рекомендуем вам использовать опции On New Record или On Post, поскольку они лучше укладываются в общую идеологию применения IBX.

Если мы получаем значение первичного ключа до отправки на сервер, то IBDataSet1 сумеет корректно выполнить RefreshSQL. Если же мы будем использовать для генерации триггер, то IBDataSet1 не сможет подставить правильное значение в RefreshSQL, поскольку значения полей в условии WHERE еще неизвестны.

Механизм master-detail

Механизм мастер-деталь часто используется в приложениях для работы с базами данных, поскольку именно он позволяет нам легко связывать данные из разных таблиц, полученных в результате нормализации базы.

Добавим на нашу форму новые компоненты (рис. 2.12).

```
IBDataSet2: TIBDataSet;
DataSource2: TdataSource;
DBGrid2: TDBGrid;
```

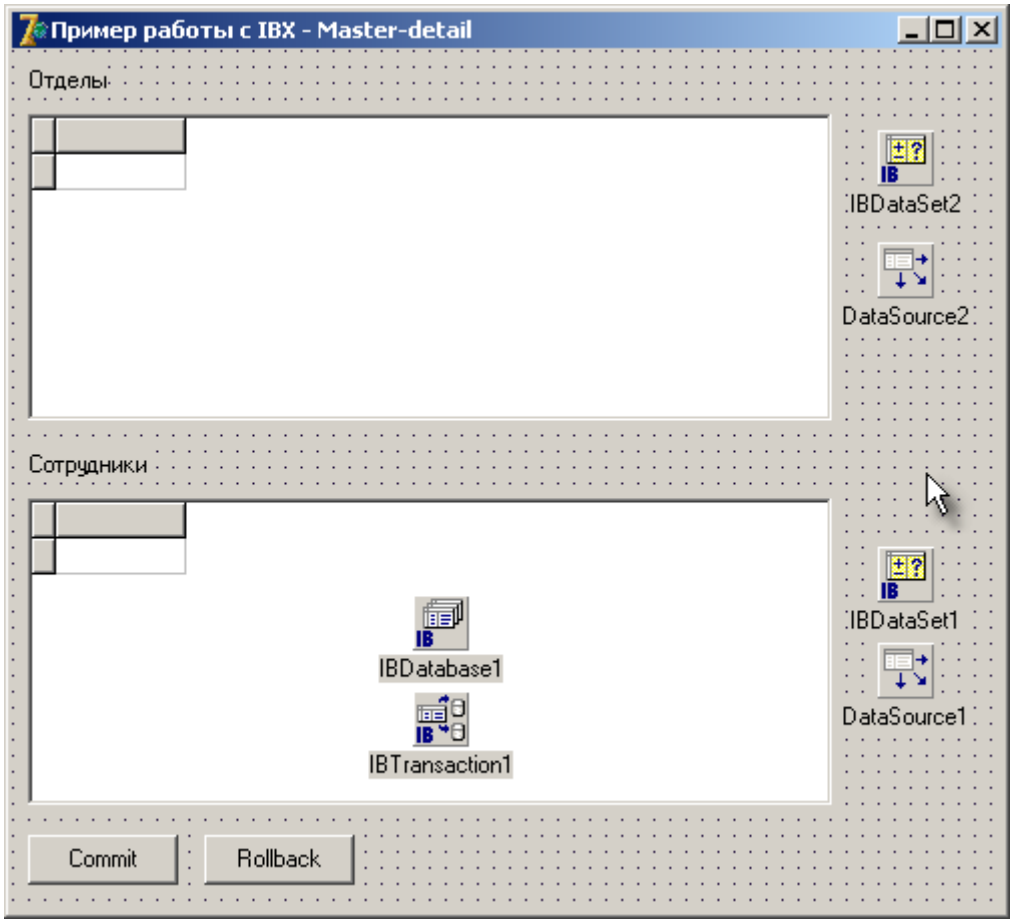


Рис. 2.12. Разработка связи мастер-деталь

Свяжем DataSource2 с IBDataSet2, а DBGrid2 с DataSource2. Укажем следующие запросы для IBDataSet2:

SelectSQL:

```
SELECT BUDGET, DEPARTMENT, DEPT_NO,  HEAD_DEPT, LOCATION,
MNGR_NO, PHONE_NO
FROM DEPARTMENT
```

InsertSQL:

```
INSERT INTO DEPARTMENT
(BUDGET, DEPARTMENT, DEPT_NO, HEAD_DEPT, LOCATION, MNGR_NO,
PHONE_NO)
VALUES
  (:BUDGET, :DEPARTMENT, :DEPT_NO, :HEAD_DEPT, :LOCATION,
:MNGR_NO, :PHONE_NO)
```

DeleteSQL:

```
DELETE FROM DEPARTMENT
WHERE  DEPT_NO = :OLD_DEPT_NO
```

ModifySQL:

```
UPDATE DEPARTMENT
SET
  BUDGET = :BUDGET,
  DEPARTMENT = :DEPARTMENT,
  DEPT_NO = :DEPT_NO,
  HEAD_DEPT = :HEAD_DEPT,
  LOCATION = :LOCATION,
  MNGR_NO = :MNGR_NO,
  PHONE_NO = :PHONE_NO
WHERE
  DEPT_NO = :OLD_DEPT_NO
```

RefreshSQL:

```
SELECT
  DEPT_NO,
  DEPARTMENT,
  HEAD_DEPT,
  MNGR_NO,
  BUDGET,
  LOCATION,
  PHONE_NO,
  DEPT_NO1
FROM DEPARTMENT
WHERE
  DEPT_NO = :DEPT_NO
```

Очевидно, что их можно сгенерировать при помощи DataSet Editor, как это было описано выше.

Теперь необходимо внести изменения в IBDataSet1. Во-первых, нужно задать свойство IBDataSet1.DataSource равным DataSource2. Во-вторых, надо изменить IBDataSet1.SelectSQL:

```
SELECT DEPT_NO, EMP_NO, FIRST_NAME, FULL_NAME, HIRE_DATE,
JOB_CODE, JOB_COUNTRY, JOB_GRADE, LAST_NAME, PHONE_EXT, SALARY
FROM EMPLOYEE
WHERE DEPT_NO = :DEPT_NO
```

Значение параметра :DEPT_NO будет автоматически браться из одноименного поля IBDataSet2: DEPT_NO.

Запустите приложение, и вы увидите, что при перемещении по верхней таблице в нижней будут отображаться только сотрудники текущего отдела (рис. 2.13).

Еще одна особенность заключается в том, чтобы настроить IBDataSet1 таким образом, что при добавлении новой записи о работнике автоматически подставлялся номер текущего отдела из IBDataSet2. Для этого напишем обработчик OnNewRecord у IBDataSet1:

```
procedure TForm1.IBDataSet1NewRecord(DataSet: TDataSet);
begin
```

```
  DataSet1.FieldByName('DEPT_NO').AsInteger := IB-
DataSet2.FieldByName('DEPT_NO').AsInteger;
end;
```

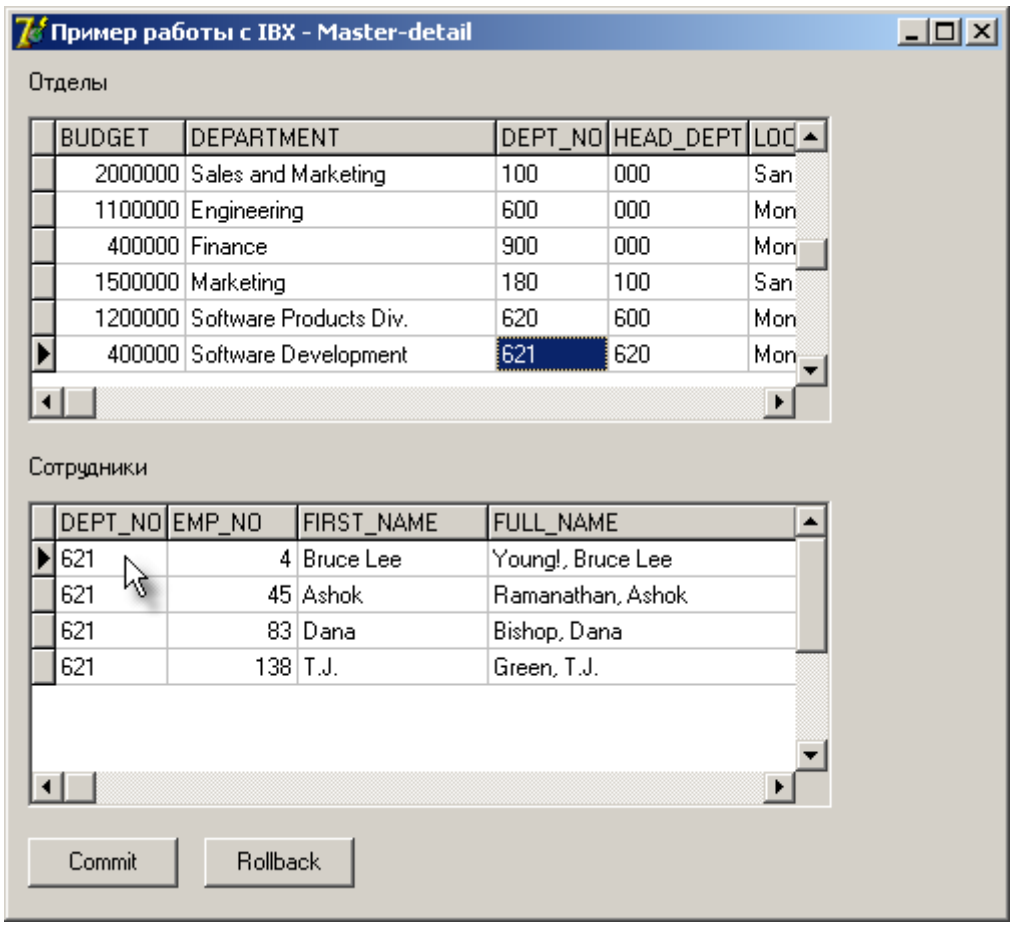


Рис. 2.13. Запущенное приложение со связью мастер-деталь

Теперь мы можем смело запускать приложение и редактировать записи в обеих таблицах.

> Хочется отметить, что в соответствующей главе, посвященной механизму мастер-деталь в FIBPlus, описаны дополнительные возможности компонентов, связанные с оптимизацией и "тонкой" настройкой связей мастер-деталь, которые отсутствуют в IBX. Возможно, что, ознакомившись с этим описанием, вы реализуете в своем приложении нечто подобное и при помощи IBX.

Также хочется отметить, что компоненты IBX предоставляют все необходимые базовые возможности для создания приложений практически любого уровня сложности.

Важно лишь понимать, какой компонент желательно применять в конкретной ситуации. Например, для массовых вставок записей в базу данных, если нет нужды отображать вставляемые записи в пользовательских компонентах, желательно использовать IBSQL, которые не буферизуют записи в отличие от TIBDataSet и его потомков.

При использовании для этой же операции TIBDataSet при большом количестве записей (порядка нескольких миллионов) приложение может начать работать на порядки медленнее.

Важно отметить, что работа с большим количеством записей для пользователя редко бывает удобной. Согласитесь, не так часто можно найти человека, который смог бы удержать в памяти информацию из нескольких сотен записей.

Обычно пользователи все-таки применяют поиск, чтобы найти нужные данные, и задача разработчика, помимо прочего, состоит в том, чтобы формулировать условия в SQL-запросах максимально точно и иметь в итоге лишь небольшой набор данных. Во-первых, это резко снизит лишний сетевой трафик, а во-вторых, снизит размер памяти, занимаемой данными в буфере TIBDataSet, за счет чего также увеличится скорость работы TIBDataSet.

Управление транзакциями тоже зависит только от вас и требований вашей задачи. Не существует единственно правильного способа распределения компонентов TIBTransaction. Некоторые предпочитают все делать в контексте одной общей транзакции, вовремя вызывая Commit или Rollback; другие предпочитают для каждого TIBDataSet иметь отдельную транзакцию, следя за правильным уровнем изоляции. Только вы можете решить, какие транзакции нужны для правильной работы ваших запросов.

Заключение

Данная глава не претендует на звание учебника по IBX, поскольку дает лишь самые основные факты и описывает только наиболее важные, как нам кажется, особенности компонентов. Однако мы надеемся, что этот материал даст вам необходимый базис для дальнейшего самостоятельного изучения.

Также заметим, что в главе по FIBPlus вы сможете найти многие вещи, которые окажутся полезными и при работе с IBX. Например, вы вполне сможете использовать материалы, описывающие работу с генератором отчетов FastReport, работу с BLOB-полями, локальную фильтрацию и обработку событий (event alerts).

Что такое FIBPlus?

Предыдущие главы книги содержали описание самого сервера InterBase, но для практического его использования необходимо разрабатывать клиентские программы. Само название технологии Client/Server говорит о наличие двух сторон: сервера, который в нашем случае обслуживает базу данных, и клиента, который является специализированной программой, взаимодействующей с сервером InterBase.

Для написания клиента, или клиентской части, вы в принципе можете использовать практически любые среды разработки, которые позволяют вызывать функции из внешней DLL-библиотеки. Суть взаимодействия с InterBase состоит в том, что ваша программа вызывает некоторые функции InterBase API для выполнения тех или иных запросов к базе данных. InterBase API заключено в специальной библиотеке gds32.dll. Таким образом, вы имеете возможность писать свои программы на чем угодно, обращаясь к функциям gds32.dll. Тем не менее, такой низкоуровневый подход неудобен для создания прикладных программ, в которых основное внимание уделяется обработке данных на достаточно абстрактном уровне. Если разработчик вынужден каждый раз заботиться, прежде всего, о том, чтобы правильно сформировать внутренние структуры для вызова функций gds32.dll, то процесс разработки значительно усложняется и затягивается.

Этого можно избежать, если вы используете для разработки клиентской части Borland Delphi, Borland C++ Builder или Borland Kylix. Идеология этих продуктов поощряет создание специализированных наборов компонент для решения определенных системных задач, облегчая, таким образом, жизнь прикладного разработчика.

Devrace FIBPlus – это библиотека компонент для Delphi, C++ Builder и Kylix, предназначенных для работы с InterBase и Firebird. Компоненты используют прямой InterBase API для взаимодействия с сервером, поэтому для нормальной работы программ, написанных с использованием FIBPlus, не нужно ничего, кроме наличия gds32.dll. Кроме того, использование прямого API дает клиентским приложениям максимальную производительность, гибкость и возможность использования "специальных" уникальных вещей, реализованных в InterBase. В частности, можно упомянуть array-поля, которые невозможно использовать при работе со стандартными Delphi компонентами TTable, TQuery, взаимодействующими с InterBase через Borland Database Engine (BDE).

Попробуем кратко перечислить основные возможности и преимущества FIBPlus:

- Полная совместимость со Borland Delphi 3-7, C++ Builder 3-6 и Kylix 3.
- Полная совместимость с существующими версиями InterBase 4.x-7.0, Firebird, Yaffil в настоящий момент, а также поддержка всех версий InterBase и его основных клонов в будущем.
- Поддержка всех особенностей и специальных возможностей InterBase: event alerters, array-полей, blob-полей.
- Встроенные макросы в SQL-выражениях, позволяющие использовать параметры-подстановки даже в там, где это не позволяет сам InterBase.

- Удивительно гибкий механизм работы с master-detail запросами, позволяющий свести "ручное" кодирование практически к нулю и при этом иметь возможность гибко настраивать любые связки master-detail без написания обработчиков событий, улучшая общую производительность приложения за счет снижения лишнего сетевого трафика.
- Улучшенная производительность при выполнении запросов со средним и небольшим количеством результирующих записей (т. е. , для 90-95% всех запросов в обычных приложениях!).
- Гибкое управление транзакциями. FIBPlus, которое позволяет, с одной стороны, явно управлять каждой транзакцией, а с другой стороны автоматизирует большинство операций, автоматически запуская транзакции, и подтверждая их в режиме AutoCommit. Кроме этого, FIBPlus реализует уникальный механизм работы одного компонента в контексте двух транзакций: читающей и пишущей, что позволяет совершенно избегать DEADLOCK при работе приложения.
- Реализация и поддержка дополнительных возможностей, например, эмуляция Boolean-полей, отсутствующих в явном виде в InterBase. Качественная поддержка режима CachedUpdates позволяет использовать FIBPlus в приложениях с нестабильными удаленными подключениями к базе данных (например, посредством dial-up). Используя CachedUpdates, вы можете подключиться к базе данных, выбрать данные, отключиться, изменить данные и после повторного подключения (или восстановления подключения), вернуть изменения на сервер. Данная возможность реализована только в FIBPlus, поскольку, например, в IBX даже при режиме CachedUpdates необходимо, тем не менее, иметь активные транзакции и подключение к серверу.
- Поддержка локальной сортировки и локальной фильтрации данных (без дополнительных запросов на сервер).
- Автоматическая генерация модифицирующих запросов в run-time и design-time. Уникальная возможность автоматически генерировать модифицирующие запросы, в которых участвуют не все поля записи, а только те, значения которых реально были изменены, что позволяет снизить сетевой траффик приложения.
- Аккуратная поддержка auto-increment полей, не требующая переоткрытия запроса после вставки новой записи, чтобы получить значения полей, измененных триггерами.
- Возможность централизованной обработки ошибок и исключений при помощи компонента TpFIBErrorHandler.
- Уникальная возможность создания цепочек модифицирующих запросов при помощи компонента TpFIBUpdateObject. Компонент позволяет реализовывать без ручного кодирования выполнение нескольких автоматических модифицирующих действий. Причем, эти модифицирующие запросы автоматически получают параметры от TpFIBDataSet и могут работать в рамках разных транзакций и даже разных подключений к базам данных. Управляя только свойствами TpFIBUpdateObject, разработчик может активировать или деактивировать нужные ему цепочки запросов. Все это превращает

TpFIBUpdateObject в некое подобие триггера, реализованного в клиентской части приложения.

- Полная поддержка Services API, что позволяет включать в приложения функции удаленного администрирования баз данных и самого сервера.

Компоненты FIBPlus построены таким образом, чтобы их можно было использовать со всеми стандартными визуальными db-компонентами и сторонними продуктами, поддерживающими стандарт TDataSet-TDataSource. Таким образом, разработчик, выбравший FIBPlus для доступа к InterBase, не ограничен в выборе сторонних компонент для отображения данных, или печати отчетов (мы покажем, в частности, как использовать FIBPlus в связке с FastReport).

Для любознательного читателя, возможно, будет также интересно узнать некоторые факты из истории появления и развития FIBPlus. В 1998 году независимый разработчик Грегори Дилтц создал минимальный набор компонентов для работы с InterBase API, назвав его FreeIBComponents (FIBC). Уже с 1999 года одесский программист Сергей Бузаджи стал дописывать эту бесплатную библиотеку своими более функциональными компонентами, назвав эти дополнения FIBPlus. Тогда же корпорация Borland выпустила Delphi 5, включив в нее компоненты InterBase Express (IBX), полностью основанные на коде FIBC. Сам Грегори Дилтц отказался от продолжения развития своей библиотеки. К этому времени изменения в FIBC, сделанные Сергеем Бузаджи, были столь значительны и затрагивали практически весь первоначальный код Грегори Дилтца, что было решено развивать FIBPlus как отдельный проект, в рамках условий лицензионного соглашения Грегори Дилтца. С этого времени поддержкой и развитием библиотеки занимается компания Devrace, Сергей Бузаджи является ведущим разработчиком и идеологическим руководителем данного продукта. На текущий момент библиотека полностью совместима с Borland Delphi 3-7, Borland C++ Builder 3-5, Borland Kylix 3 и поддерживает Borland InterBase 4-7.0, а также все версии Firebird и Yaffil.

> FIBPlus является коммерческим продуктом, однако компания Devrace ввела специальную цену для стран бывшего Союза, которая в десятки раз меньше цены для "западных" пользователей.Вы можете найти всю информацию о FIBPlus на сайтах: http://www.devrace.com, http://www.fibplus.net.

Общее описание компонент, включенных в состав FIBPlus

 TpFIBDatabase – предназначен для подключения к базе данных, позволяет получать статистическую информацию о базе, свойствах базы данных и так далее. Основные методы: Open, Close.

 TpFIBTransaction – предназначен для явного управления транзакцией. Основные методы: StartTransaction, Commit, Rollback, CommitRetaining, RollbackRetaining.

 TpFIBDataSet – предназначен для получения и редактирования данных, является потомком стандартного класса TDataSet и полностью совместим со всеми визуальными компонентами. Основные методы: Prepare, Open, Close, Insert, Append, Edit, Delete, Refresh.

 TpFIBQuery – предназначен для простого выполнения запросов, позволяет также получать данные из базы данных. В отличие от TpFIBDataSet , компонент не содержит локального буфера для хранения всех полученных записей, и не совместим с визуальными компонентами. Основные методы: Prepare, ExecQuery, Close

 TpFIBStoredProc – потомок TpFIBQuery, предназначен для выполнения хранимых процедур. Не рекомендуется для вызова процедур, возвращающих данные. Фактически, компонент является удобной "оберткой" вокруг команды EXECUTE PROCEDURE.

 TpFIBUpdateObject – предназначен для автоматического выполнения дополнительных модифицирующих действий совместно с TpFIBDataSet.

 TDataSetContainer – предназначен для централизации обработки типичных обработчиков событий TpFIBDataSet и внутренней нотификации, что иногда позволяет сильно упростить код.

 TpFIBSQLMonitor – предназначен для отслеживания всех запросов, которые выполняются при помощи компонентов FIBPlus. Позволяет также отслеживать запросы из параллельных приложений, также написанных на основе FIBPlus.

 TpFIBErrorHandler - предназначен для централизованной обработки исключительных ситуаций, возникших при работе с функциями InterBase API. Позволяет обрабатывать потерю подключения к базе данных, инициализацию пользовательских исключений в самой базе данных.

 SuperIBAlerter – предназначен для регистрации и получения событий (event alerters) InterBase.

 TIBConfigService – предназначен для настройки параметров базы данных.

 TIBBackupService – предназначен для создания резервных копий (backup) баз данных.

 TpFIBRestoreService – предназначен для восстановления базы данных из резервной копии.

 TpFIBValidationService – предназначен для проверки целостности базы данных и согласования внутренних данных о транзакциях.

 TpFIBStatisticalService – предназначен для получения статистики о базе данных.

 TpFIBLogService – предназначен для создания и просмотра лог-файла работы сервера.

 TpFIBSecurityService – предназначен для редактирования списка пользователей на сервере.

 TpFIBLicensingService – предназначен для добавления и удаления сертификатов, регулирующих количество и свойства клиентских подключений к серверу InterBase.

 TpFIBServerProperties – предназначен для получения информации о сервере, параметров конфигурации и так далее.

 TpFIBInstall – предназначен для установки компонента InterBase installation.

 TpFIBUnInstall – предназначен для установки компонента InterBase uninstallation.

Использование основных компонентов FIBPlus

В процессе более подробного рассмотрения вопросов использования компонентов FIBPlus мы постараемся создать очень простое приложение, которое будет предназначено для редактирования некоторого достаточно абстрактного прайс-листа. Этот пример позволит рассмотреть все основные аспекты работы с FIBPlus, с которыми так или иначе сталкивается практически каждый прикладной разработчик.

Наше приложение должно позволять редактировать категории товаров, список товаров для каждой категории и печатать прайс-лист. Нам понадобятся две таблицы: для категорий товаров и для списка товаров:

```
CREATE TABLE "Categories" (
    "Id" INTEGER NOT NULL,
    "Name" VARCHAR (50) character set WIN1251 collate PXW_CYRL,
    "GoodsCount" INTEGER);

/* Unique keys definition */

ALTER TABLE "Categories" ADD CONSTRAINT "PK_Categories" PRIMARY
KEY ("Id");

SET TERM ^ ;

/* Trigger: "AI_Categories_Id" */
CREATE TRIGGER "AI_Categories_Id" FOR "Categories" ACTIVE
BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW."Id" IS NULL) THEN
      NEW."Id" = GEN_ID("Categories_Id_GEN", 1);
END
^
SET TERM ; ^

CREATE TABLE "Goods" (
    "Id" INTEGER NOT NULL,
    "Name" VARCHAR (150) character set WIN1251 collate
PXW_CYRL,
    "Price" DOUBLE PRECISION,
    "IdCategory" INTEGER NOT NULL);

/* Unique keys definition */

ALTER TABLE "Goods" ADD CONSTRAINT "PK_Goods" PRIMARY KEY
("Id");

/* Foreign keys definition */
```

```
ALTER TABLE "Goods" ADD  CONSTRAINT FK_GOODS FOREIGN KEY
("IdCategory") REFERENCES "Categories" ("Id") ON DELETE CASCADE
ON UPDATE CASCADE;

SET TERM ^ ;

/* Trigger: "AI_Goods_Id" */
CREATE TRIGGER "AI_Goods_Id" FOR "Goods" ACTIVE
BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW."Id" IS NULL) THEN
      NEW."Id" = GEN_ID("Goods_Id_GEN", 1);
END
^
```

Очевидна связь master-detail между таблицами "Categories" и "Goods".

> Обратите внимание, что мы с самого начала и в дальнейшем будем использовать особенности SQLDialect 3, который доступен в InterBase начиная с версии 6.0
>
> Для печати прайс-листа мы будем использовать генератор отчетов FastReport (http://www.fastreport.ru).

Подключение к базе данных, выполнение простых запросов

Рассмотрим с самого начала создание приложения, при помощи которого мы сможем редактировать прайс-лист. Необходимо поместить на форме основной компонент, который позволяет подключаться к базе данных InterBase (TpFIBDatabase) и вызвать редактор этого компонента (рис. 2.14 и 2.15).

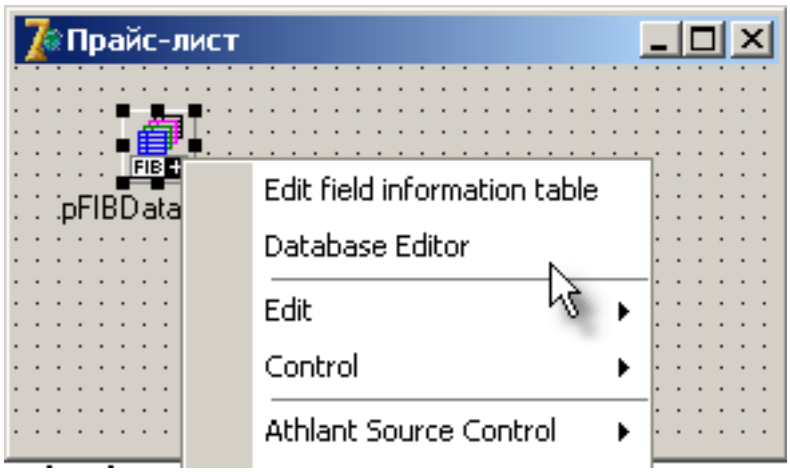


Рис. 2.14. Вызов редактора TpFIBDataBase

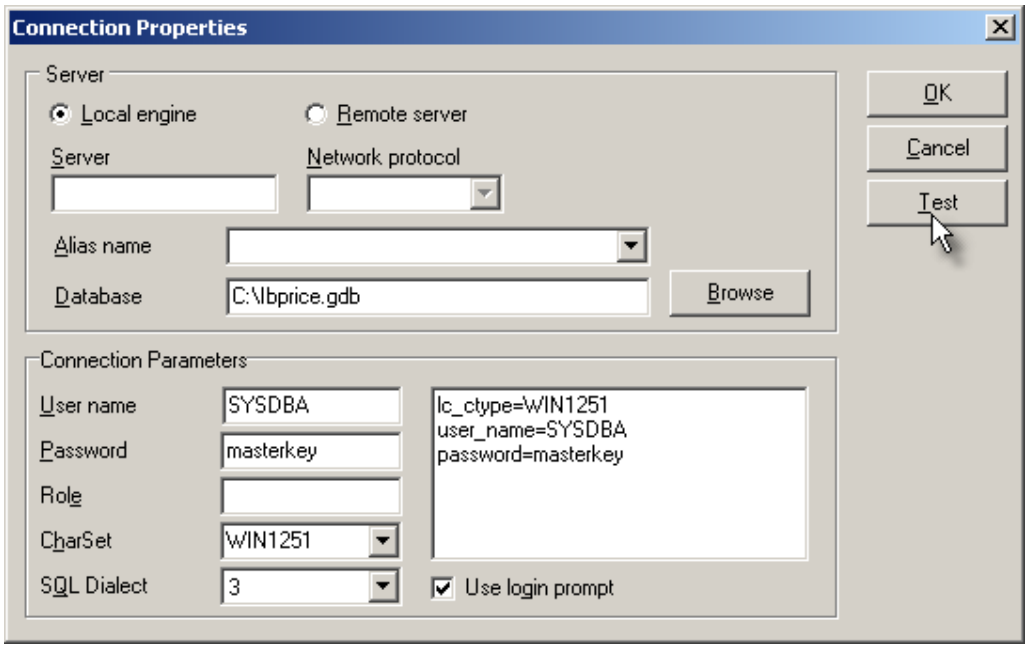


Рис. 2.15. Вид редактора компонента TpFIBDataBase

Для подключения к базе как минимум необходимо указать путь (в данном примере это путь к локальному файлу), имя пользователя и пароль. Вы можете проверить правильность заданных параметров, нажав на кнопку Test. Мы также можем задать параметры подключения к базе в run-time, получив путь к базе данных из ini-файла:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  with TiniFile.Create('ib_price.ini') do begin
     pFIBDatabase1.DBName := ReadString('Options', 'DBPath',
'C:\IBPRICE.GDB');
    Free;
  end;
  pFIBDatabase1.Open;
end;
```

Компонент TpFIBDatabase можно также использовать для выполнения запросов к базе данных, которые не возвращают в результате набора данных. Для этого существуют такие методы:

```
function Execute(const SQL: string): boolean;
function QueryValue(const aSQL: string; Field-
No:integer):Variant;
function QueryValueAsStr(const aSQL: string; Field-
No:integer):String;
```

Например, мы можем выяснить количество категорий товаров, выполнив простой запрос:

```
ShowMessage(pFIBDatabase1.QueryValueAsStr('select count("Id")
from "Categories"', 0));
```

Управление транзакциями

Фактически любые действия с данными должны происходить в контексте той или иной транзакции. Управление транзакциями в FIBPlus осуществляется при помощи компонентов класса TpFIBTransaction. Все транзакции в FIBPlus являются "явными" (explicit) и запускаются при помощи метода StartTransaction. Тем не менее, чтобы вы могли избежать лишнего кодирования, TpFIBDataSet и TpFIBQuery самостоятельно запускают транзакции, если установлен ключ poStartTrasaction в свойстве Options. Завершать транзакцию в любом случае необходимо явным образом при помощи вызова соответствующих методов: Commit, Rollback, CommitRetaining и RollbackRetaining.

> Метод CommitRetaining появился в InterBase только начиная с версии 5.1, а метод RollbackRetaining – только в версии 6.0.

Планируя внутреннюю организацию транзакций в приложении, необходимо помнить про уровень изоляции транзакций. Можно указывать уровень изоляции транзакции явным образом при помощи соответствующих констант InterBase в свойстве TRParams или используя заранее заданные уровни изоляции при помощи свойства TPBMode. По умолчанию любой компонент TpTransaction, помещенный на форму, имеет уровень изоляции "Read Committed".

FIBPlus также позволяет создавать и запоминать в системном реестре пользовательские уровни изоляции (свойство UserKindTransaction). Необходимо вызвать редактор компонента, нажав на нем правой кнопкой мыши, и выбрать пункт "Edit transaction params" (рис. 2.16).

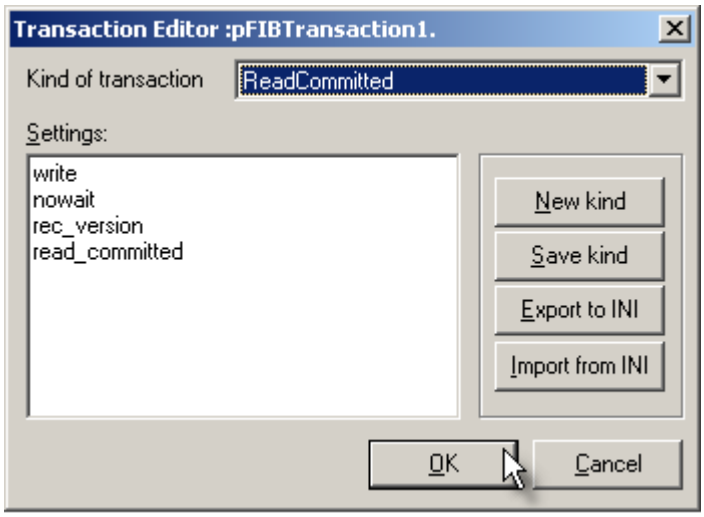


Рис. 2.16. Редактор компонента TpFIBTransaction

После нажатия на кнопку New Kind нужно указать название для набора констант и перечислить необходимые константы в поле Settings. Теперь нужно сохранить константы нажатием кнопки Save kind. Описания всех констант вы можете узнать в документации к InterBase. В дальнейшем, вы можете использовать

собственные созданные наборы констант, выбирая названия из списка в свойстве UserKindTransaction.

Закрытие транзакции также имеет ряд особенностей, которые необходимо иметь в виду разработчикам. Если вы закрываете транзакцию вызовом методов Commit или Rollback, то все активные запросы, которые работают в контексте этой транзакции, будут также закрыты. Такое поведение непривычно для разработчиков, ранее использовавших в своих приложениях BDE, где подтверждение транзакции оставляло курсоры открытыми. Нужно подчеркнуть, что механизм, реализованный в BDE, является лишь эмуляцией. То есть фактически завершение транзакции просто вызывало невидимое автоматическое "переоткрытие" всех активных запросов. Кроме того, важно иметь в виду, что все запросы при использовании BDE работают в контексте одной и той же транзакции, в отличие от FIBPlus, где каждый запрос может работать в рамках своей отдельно взятой транзакции.

Тем не менее если вы не хотите, чтобы подтверждение транзакции вызывало закрытие всех активных TpFIBDataSet, то вы можете использовать метод CommitRetaining. Этот метод подтверждает транзакцию и автоматически запускает новую с теми же самыми параметрами, не закрывая при этом пользовательских курсоров (запросов, выбирающих данные).

> Данный метод содержал ошибку реализации, поэтому не рекомендуется слишком часто вызывать CommitRetaining для версий InterBase меньше чем 6.5.

То же самое касается и метода RollbackRetaining, который отменяет изменения, сделанные в контексте транзакции и запускает новую.

> Данный метод появился в InterBase версии 6.0.

Использование стандартных визуальных db-компонентов совместно с FIBPlus

Следующий этап – получение данных из базы данных и отображение этих данных при помощи визуальных компонентов (рис. 2.17).

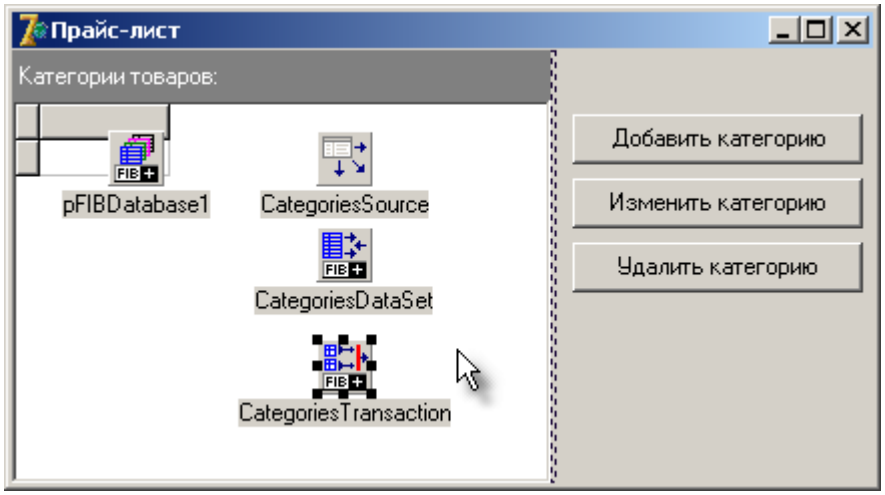


Рис. 2.17. Подключение визуальных компонентов к TpFIBDataSet

Для этого нам необходимы два компонента: TpFIBDataSet и TpFIBTransaction. TpFIBDataSet является потомком класса TDataSet и полностью совместим со всеми стандартными визуальными компонентами. Для связки TpFIBDataSet с визуальным компонентом CategoriesGrid (TDBGrid) компонент CategoriesDataSource подключен к CategoriesDataSet. TpFIBTransaction, как уже было сказано, предназначен для управления транзакцией.

Необходимо указать уровень изоляции транзакции при помощи свойства TPBMode, а также "подключить" CategoriesTransaction к pFIBDatabase1 (2.18).

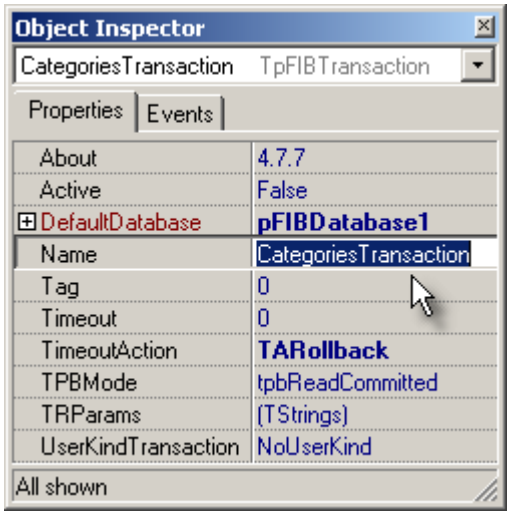


Рис. 2.18. Подключение транзакции к компоненту pFIBDataBase1

Аналогично необходимо указать свойства Database и Transaction для компонента CategoriesDataSet (рис. 2.19).

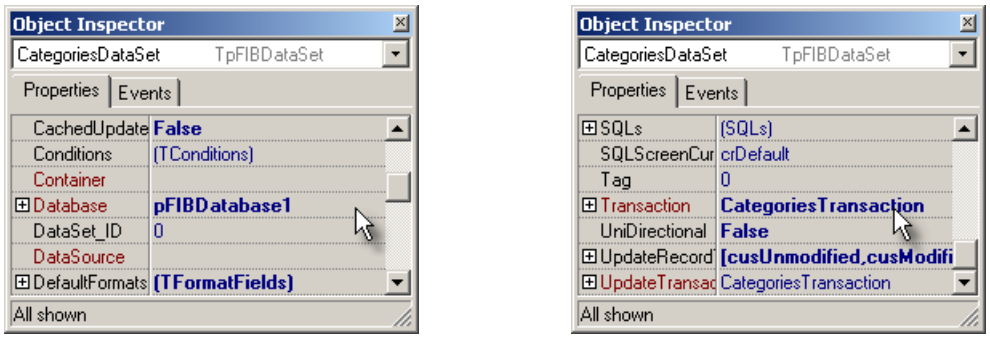


Рис. 2.19. Подключение компонента CategoriesDataSet к pFIBDataBase1 и транзакции CategoriesTransaction

Теперь мы можем указать необходимые запросы к базе данных для получения и изменения данных в таблице. Нужно написать запрос для получения дан-

ных, используя свойство SelectSQL. В нашем случае это будет очень простой запрос:

```
SELECT * FROM "Categories"
```

Теперь изменим немного код в процедуре:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  with TiniFile.Create('ib_price.ini') do begin
     pFIBDatabase1.DBName := ReadString('Options', 'DBPath',
'C:\IBPRICE.GDB');
    Free;
  end;
  pFIBDatabase1.Open;

  CategoriesDataSet.Open;
end;
```

В свойстве DataSet у компонента CategoriesSource (TDataSource) мы укажем CategoriesDataSet, а в свойстве DataSource у компонента CategoriesGrid (TDBGrid) укажем CategoriesSource. Теперь все данные, полученные в результате запроса, будут отображены в CategoriesGrid. Нам не нужно явным образом запускать транзакцию перед открытием запроса, поскольку по умолчанию свойство CategoriesDataSet.Options содержит ключ poStartTransaction и компонент сам запускает транзакцию. После запуска приложения и открытия базы данных мы увидим следующий результат (рис. 2.20):

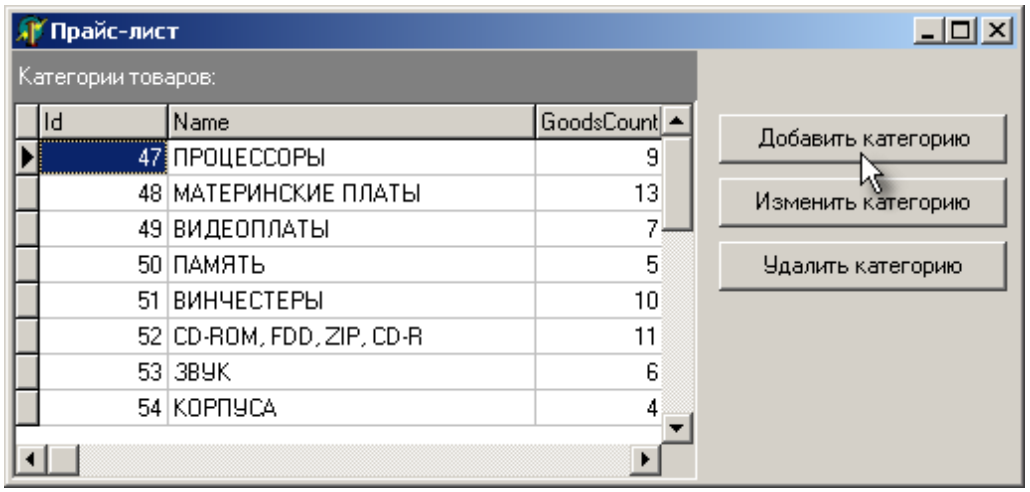


Рис. 2.20. Вид запущенного приложения

Вместо CategoriesGrid мы могли использовать любой визуальный db-компонент, который используется в связке с TDataSource. Это мог быть TDBEdit, TDBLabel, TDBMemo и т. д., а также любые сторонние компоненты, которые придерживаются данного стандарта (TDBGridEh, например).

Как сделать запрос редактируемым? Автоматическая генерация модифицирующих запросов в design-time и run-time

Если вы попробуете исправить какие-то значения в CategoriesGrid, то увидите, что в текущем состоянии мы имеем read-only-запрос. Для того чтобы наш запрос стал редактируемым, или "живым", нам надо написать несколько дополнительных запросов, которые будут автоматически выполнятся компонентом CategoriesDataSet при редактировании данных в CategoriesGrid. Необходимо заполнить свойства (рис. 2.21).

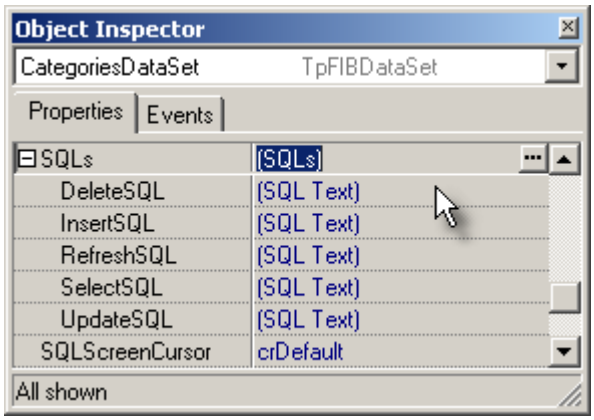


Рис. 2.21. Свойства CategoriesDataSet с запросами для чтения и модификации данных

FIBPlus включает специальный design-time-редактор для редактирования и генерации модифицирующих запросов. Вы можете вызвать "Generator SQLs" из контекстного меню, если нажмете правой кнопкой мыши на компоненте CategoriesDataSet (рис. 2.22).

Для генерации модифицирующих запросов необходимо сначала указать основную таблицу в списке. Если в запросе участвует только одна таблица, то она будет выбрана в списке автоматически. После этого необходимо нажать кнопку Get Table Fields для заполнения списков "Key Fields" и "Update Fields". В первом списке нужно выделить те поля, которые будут вставлены в условие WHERE во всех модифицирующих запросах. В списке "Update Fields" необходимо выделить те поля, которые мы хотим редактировать. По умолчанию выделены все поля таблицы "Categories". Теперь достаточно нажать кнопку Generate SQLs и получить автоматически все модифицирующие запросы, которые вы можете увидеть на закладке SQLs. Например, мы получим следующий запрос для свойства InsertSQL:

```
INSERT INTO "Categories"(
    "Id",
    "Name",
    "GoodsCount"
)
```

```
VALUES(
     :"Id",
     :"Name",
     :"GoodsCount"
)
```

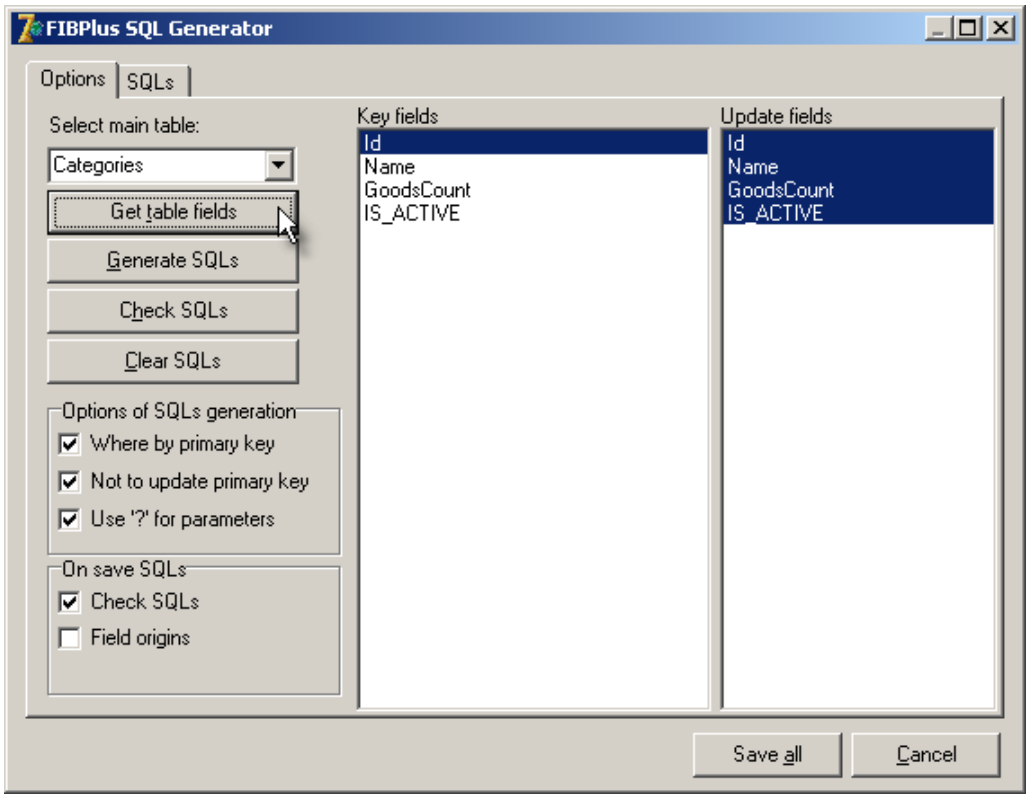


Рис. 2.22. Редактор компонента TpFIBDataSet, предназначенный для генерации модифицирующих запросов

Обратите внимание также на запрос для свойства RefreshSQL:

```
select * from "Categories"
 WHERE
     (
             "Categories"."Id" = :"OLD_Id"
     )
```

Это такой же выбирающий запрос, как и запрос в свойстве SelectSQL, но возвращать он должен только одну запись – текущую. Теперь, после создания всех запросов, когда пользователь будет редактировать запись в CategoriesGrid, CategoriesDataSet автоматически заполнит значения параметров : "Id", : "Name" и т. д., теми значениями полей, которые наберет пользователь. После выполнения метода Post (который, в частности, вызывается при нажатии клавиши Enter в CategoriesGrid) CategoriesDataSet выполнит соответствующий модифицирую-

щий запрос. После редактирования записи будет выполнен запрос из свойства UpdateSQL, при вставке новой записи – из InsertSQL, а при удалении записи – из DeleteSQL. После выполнения любого модифицирующего запроса (кроме удаления) CategoriesDataSet автоматически выполняет запрос из RefreshSQL, подставляя в качестве условия текущие значения параметров. Это позволяет "узнать" значения всех полей текущей записи, если они были изменены при помощи триггеров базы данных.

Несколько слов о специальных префиксах для параметров, которые используются в FIBPlus.

Префикс "OLD_" фактически означает текущее значение одноименного поля или же предыдущее значение поля. Термин "предыдущее значение" имеет смысл в запросе на удаление. Во всех остальных случаях, когда нужны актуальные значения полей, можно обходиться без данного префикса.

Префикс "NEW_" означает новое значение поля и фактически имеет смысл только в запросах на изменение и вставку. Однако новые значения полей в этих ситуациях также являются и текущими, а значит, префикс "NEW_" можно опускать.

Существует также специальный префикс "MAS_", но мы доберемся до него несколько позже.

Поскольку мы сгенерировали все необходимые модифицирующие запросы, то уже можем запустить наше приложение, добавить пару категорий, изменить их названия и даже целиком удалить их.

Тем не менее генерация модифицирующих запросов в design-time не всегда удобна. FIBPlus позволяет разработчику автоматически генерировать нужные запросы во время работы приложения, основываясь на запросе из свойства SelectSQL. Для этого нужно использовать свойство AutoUpdateOptions (рис. 2.23).

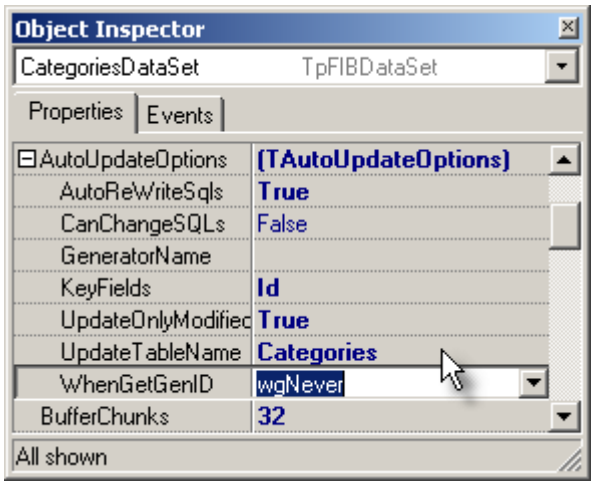


Рис. 2.23. Комплексное свойство AutoUpdateOptions

Необходимо "включить" ключ "AutoReWriteSqls", указать ключевое поле, которое будет использовано в условиях WHERE (в нашем случае, это поле "Id")

и указать название модифицируемой таблицы ("Categories"), поскольку в реальности запросы могут быть многотабличными. Теперь свойства InsertSQL, DeleteSQL, UpdateSQL и RefreshSQL будут генерироваться автоматически при открытии CategoriesDataSet. То есть фактически мы имеем генератор запросов в run-time, что бывает очень удобно.

Есть еще один момент, связанный с генерацией модифицирующих запросов. Например, если мы редактируем только название категории, то после нажатия на Enter, на сервер будет отправлен запрос, в котором изменены два поля: и "Name" и "GoodsCount". Фактически же мы изменяли только одно поле – "Name". В итоге если запись таблицы содержит значительный объем данных (например, длинные строковые поля), то при малейшем исправлении даже одного поля на сервер будут "уходить" все данные. При многопользовательской работе это создаст дополнительный лишний трафик. Данной ситуации можно избежать, но для этого необходимо, чтобы приложение выполняло более эффективные изменяющие запросы, например запрос:

```
UPDATE "Categories" SET
    "Name" = :"Name"
WHERE
            "Id" = :"OLD_Id"
```

в котором, как вы видите, участвует только то поле, значение которого действительно было изменено. Такая возможность существует, если мы включим ключ UpdateOnlyModifiedFields (разумеется, этот ключ действует, только если мы используем режим генерации запросов в run-time). В случае редактирования только поля "Name" компонент CategoriesDataSet автоматически сгенерирует запрос, приведенный выше, и будет генерировать каждый раз новые запросы в зависимости от того, какие поля изменялись.

Поскольку теперь наш CategoriesDataSet стал модифицируемым, то мы можем написать обработчики событий OnClick у трех кнопок справа от CategoriesGrid:

```
procedure TMainForm.AddCatButtonClick(Sender: TObject);
begin
  CategoriesDataSet.Append;
end;

procedure TMainForm.EditCatButtonClick(Sender: TObject);
begin
  CategoriesDataSet.Edit;
end;

procedure TMainForm.DeleteCatButtonClick(Sender: TObject);
begin
  if MessageDlg('Вы уверены, что хотите удалить категорию "' +
   CategoriesDataSet.FieldByName('Name').AsString + '"?',
mtConfirmation,
   [mbYes, mbNo], 0) = mrYes then
     CategoriesDataSet.Delete;
end;
```

Правильный способ использования auto-increment полей в FIBPlus

В текущем состоянии наш запрос уже является редактируемым, мы можем вставлять новые записи, удалять записи существующие, а также редактировать значения полей. Тем не менее пользователь должен самостоятельно указывать значение ключевого поля "Id", чтобы каждая запись была уникальной. Разумеется, мы можем переложить заботу о генерации уникальных значений на базу данных, написав соответствующий триггер, который при вставке новой записи будет автоматически получать новое уникальное значение из некоторого генератора. Такой способ гарантирует уникальность значений, но не гарантирует правильность работы клиентского приложения.

Особенность использования генераторов при написании корректно работающих приложений на FIBPlus состоит в том, что мы должны получать новое значение генератора в приложении до того, как выполним запрос на вставку записи. Зачем это нужно? Дело в том, что, как уже было сказано, после выполнения любого модифицирующего запроса (кроме удаления) CategoriesDataSet автоматически выполняет запрос из RefreshSQL, подставляя в качестве условия текущие значения параметров. В нашем случае для подстановки надо использовать значение первичного ключа (поле "Id"). Если мы не получим его заранее, а будем генерировать его, используя триггер, то мы не сможем подставить значение параметра : "OLD_Id" в запрос RefreshSQL, а значит, не сможем перечитать измененную запись. Таким образом, если какие-то поля записи были изменены триггерами базы данных, то мы не увидим этих изменений, пока не переоткроем весь запрос целиком. Если же мы сначала получим новое значение генератора, а потом вставим это значение наравне с остальными параметрами, то затем мы сможем использовать это же значение, чтобы "перечитать" текущую запись, и будем "в курсе" актуальных значений полей без излишних переоткрываний!

TpFIBDataSet позволяет автоматически получать и вставлять значения первичного ключа, используя генератор. Для этого нам необходимо заполнить некоторые дополнительные ключи в свойстве AutoUpdateOptions (рис. 2.24).

Нужно указать название генератора Categories_Id_GEN. Свойство WhenGetGenID может принимать три возможных значения:

wgOnNewRecord – получать значение генератора сразу после подготовки буфера для новой записи;

wgBeforePost – получать значение генератора непосредственно перед отправкой новой записи на сервер;

wgNever – не использовать механизм генерации ключевых значений.

В нашем примере мы укажем значение wgOnNewRecord, чтобы сразу видеть в CategoriesGrid те значения поля "Id", которые будут получены с сервера. Теперь мы можем запустить наше приложение и отредактировать какие-либо существующие записи и даже вставить новые записи (рис. 2.25).

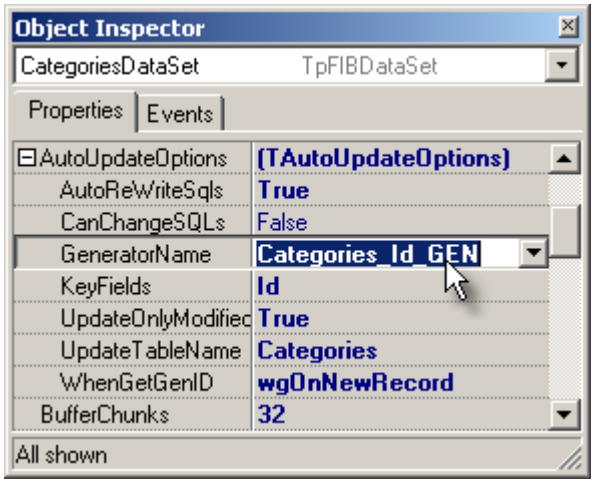


Рис. 2.24. Использование AutoUpdateOptions для генерации уникальных значений первичного ключа

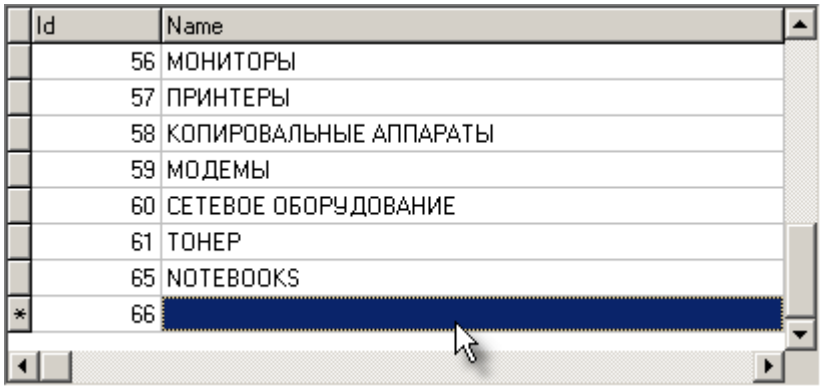


Рис. 2.25. Внешний вид "живого" запроса

Из рисунка видно, что было автоматически получено значение для поля "Id" равное 66.

Разделенные транзакции: уникальная возможность избежать Deadlock. Режим AutoCommit

CategoriesDataSet позволяет автоматически подтверждать сделанные изменения, если задать свойство AutoCommit в True. Теперь после вызова метода Post компонент CategoriesDataSet будет автоматически вызывать CategoriesTransaction.CommitRetaining, сохраняя сделанные пользователем изменения и не закрывая при этом сам запрос. Такой подход уже снижает вероятность Deadlock, который возможен при наличии длинных незакрытых транзакций, в контексте которых производились изменения данных.

Тем не менее, FIBPlus предлагает другую возможность, которая сводит вероятность возникновения Deadlock практически к нулю. TpFIBDataSet может работать одновременно в контексте двух транзакций. Одна длинная транзакция, в контексте которой данные только читаются, и другая короткая транзакция, в контексте которой выполняются все модифицирующие запросы.

Переименуем CategoriesTransaction в CategoriesReadTransaction и добавим еще один компонент CategoriesWriteTransaction. После этого зададим свойство UpdateTransaction у CategoriesDataSet в CategoriesWriteTransaction. Таким образом, теперь CategoriesDataSet подключен сразу к двум компонентам TpFIBTransaction (рис. 2.26).

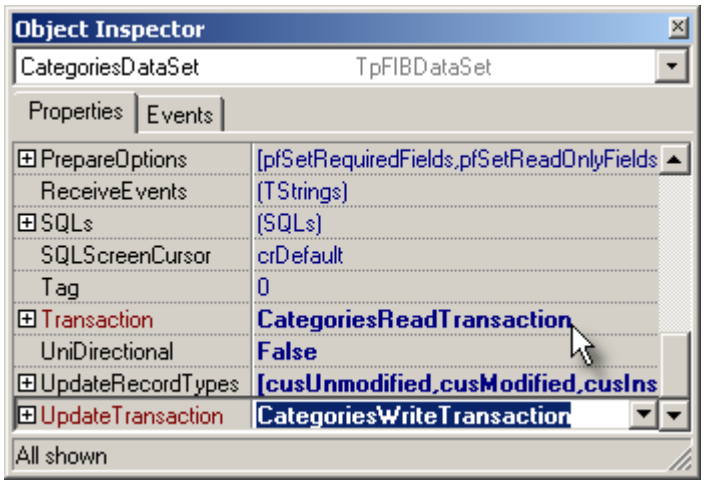


Рис. 2.26. Разделение читающей и пишущей транзакций

После вызова метода Post компонент CategoriesDataSet будет вызывать метод Commit у компонента CategoriesWriteTransaction. Однако, учитывая, что данные читаются в контексте совсем другой транзакции (CategoriesReadTransaction), это не вызовет закрытия всего CategoriesDataSet. То есть, используя FIBPlus, мы имеем настоящий режим AutoCommit, который уменьшает вероятность Deadlock и не мешает "видеть" нам актуальные значения всех записей. При использовании BDE в режиме AutoCommit вы не могли бы узнать реальные значения записей, пока не переоткроете запрос.

Кроме того, как уже упоминалось, для InterBase до версии 6.5 слишком частый вызов CommitRetaining мог привести к значительному "захвату" ресурсов сервером. При использовании механизма разделенных транзакций вы можете использовать режим AutoCommit без потерь производительности сервера для любых версий InterBase.

Механизм master-detail. Специальные опции TpFIBDatabase и TpFIBDataSet

Мы имеем возможность редактировать данные о категориях товаров и можем переходить к вопросам, связанным с построением связки master-detail. Для этого мы положим на форму дополнительные компоненты:

```
GoodsSource: TDataSource;
GoodsGrid: TDBGrid;
GoodsDataSet: TpFIBDataSet;
GoodsReadTransaction: TpFIBTransaction;
GoodsWriteTransaction: TpFIBTransaction;
AddGoodsButton: TButton;
EditGoodsButton: TButton;
DeleteGoodsButton: TButton;
```

"Свяжем" их так же, как и предыдущую группу компонентов, и напишем SelectSQL для GoodsDataSet:

```
SELECT * FROM "Goods"
WHERE "IdCategory" = :"Id"
```

Очевидно, что мы хотим выбрать при помощи detail-запроса только те товары, которые относятся к текущей категории. Значение параметра : "Id" должно браться из поля "Id" таблицы "Categories". Чтобы это происходило автоматически, необходимо задать свойство GoodsDataSet.DataSource равным CategoriesSource. Теперь сгенерируем модифицирующие запросы для GoodsDataSet так же, как мы делали это раньше для CategoriesDataSet.

После автоматической генерации мы должны внести некоторые изменения в полученные запросы. Рассмотрим, в частности, запрос для InsertSQL:

```
INSERT INTO "Goods"(
    "Id",
    "Name",
    "Price",
    "IdCategory"
)
VALUES(
    :"Id",
    :"Name",
    :"Price",
    :"IdCategory"
)
```

Очевидно, что при добавлении нового товара, мы должны задать параметр :"IdCategory" текущим значением поля "Id" из таблицы "Categories". FIBPlus позволяет делать это автоматически при помощи префикса "MAS_", о котором мы уже упоминали выше:

```
INSERT INTO "Goods"(
    "Id",
    "Name",
    "Price",
    "IdCategory"
)
VALUES(
    :"Id",
    :"Name",
    :"Price",
    :"MAS_Id"
)
```

Теперь мы указали явным образом, что значение для поля "Goods"."IdCategory" нужно брать из поля "Id" таблицы "Categories", которая является master-таблицей для таблицы "Goods". То же изменение необходимо внести в запрос UpdateSQL:

```
UPDATE "Goods" SET
    "Id" = :"Id",
    "Name" = :"Name",
    "Price" = :"Price",
    "IdCategory" = :"MAS_Id"
 WHERE
            "Id" = :"OLD_Id"
```

Теперь изменим немного код в процедуре FormCreate:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  with TiniFile.Create('ib_price.ini') do begin
     pFIBDatabase1.DBName := ReadString('Options', 'DBPath',
'C:\IBPRICE.GDB');
    Free;
  end;
  pFIBDatabase1.Open;

  CategoriesDataSet.Open;
  GoodsDataSet.Open;
end;
```

Не забудем заполнить свойства AutoUpdateOptions, чтобы GoodsDataSet автоматически генерировал значения первичного ключа (рис. 2.27).

Можно запустить приложение. Двигаясь вверх и вниз по CategoriesGrid, вы можете наблюдать, как автоматически изменяется содержимое в GoodsGrid. Связка master-detail уже работает.

FIBPlus позволяет также настраивать некоторые особенности работы механизма master-detail. В частности, как вы уже обратили внимание, в нашем примере мы открывали detail-dataset (GoodsDataSet) "вручную", при помощи явного вызова метода Open. В случае с простой связкой это нетрудно, однако если мы используем несколько цепочек master-detail или более длинные цепочки – master-detail-subdetail, то ручное открытие всех запросов может даже привести к ошибке. Вы всегда должны открывать подчиненный запрос после основного.

Чтобы избежать ненужного и совершенно очевидного кодирования, TpFIBDataSet содержит специальное свойство DetailConditions (рис. 2.28).

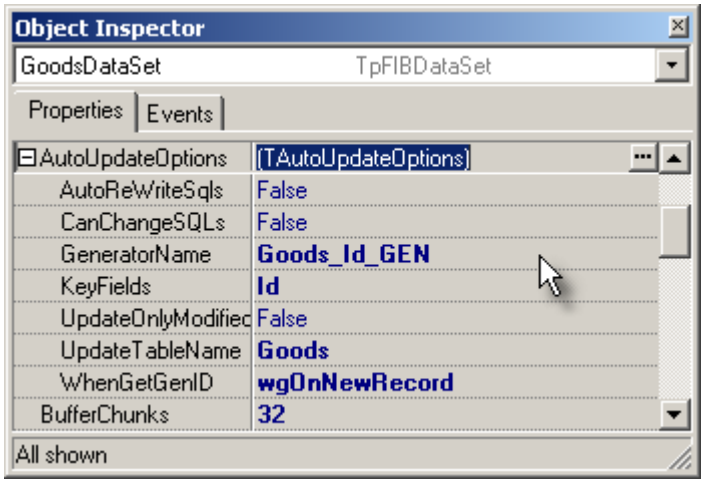


Рис. 2.27. Использование AutoUpdateOptions для генерации значений первичного ключа GoodsDataSet

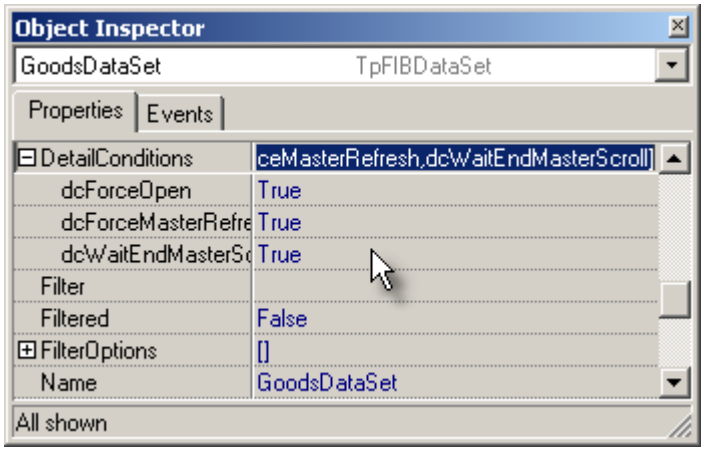


Рис. 2.28. Свойство DetailConditions

Задав ключ dcForceOpen, мы можем быть совершенно уверены, что detail-запрос будет автоматически открыт после открытия master-запроса. Теперь мы можем удалить строку GoodsDataSet.Open из метода CreateForm. Какова бы ни была глубина связок master-detail, все запросы будут открыты в нужном порядке.

Вторая важная опция – это ключ dcWaitEndMasterScroll. Представьте, что пользователь передвигается по CategoriesGrid, пытаясь найти нужную категорию товаров. При каждом движении, когда меняется текущая запись в CategoriesGrid, GoodsDataSet автоматически открывает запрос заново. Очевидно, движение по master-таблице может вызвать серию довольно "тяжелых" запро-

сов, которые значительно увеличат совершенно бесполезный сетевой трафик. На практике было нужно переоткрыть detail-запрос только один раз, когда пользователь все-таки нашел нужную ему категорию в CategoriesGrid. FIBPlus позволяет избежать ненужных запросов. Если ключ dcWaitEndMasterScroll добавлен в DetailConditions, то detail-запрос переоткрывается только после некоторой паузы (величину паузы можно регулировать, задавая значение свойства WaitEndMasterInterval).

Таким образом, когда пользователь просто передвигается по CategoriesGrid, изменение текущей записи "включает" таймер detail-запроса. Если за время паузы пользователь успевает перейти на другую запись, то таймер сбрасывается. И так до тех пор, пока пользователь не остановится на нужной записи. Только после этого detail-запрос переоткрывается, а значит, GoodsDataSet выполняет только один запрос вместо целой серии.

Третий важный ключ в DetailConditions управляет поведением master-таблицы при изменении detail-таблицы. В нашем примере в таблице "Categories" существует поле "GoodsCount", в котором содержится количество наименований товаров для текущей категории. Когда мы добавляем или удаляем наименование товара из категории, то это значение должно меняться. Очевидно, что данный механизм легко реализуется парой триггеров для таблицы "Goods":

```
CREATE TRIGGER "Goods_AI" FOR "Goods" ACTIVE
AFTER INSERT POSITION 0
AS
BEGIN
  update "Categories" set
    "GoodsCount" = (select count("Id") from "Goods" where
"IdCategory" = New."IdCategory")
  where "Id" = New."IdCategory";
END^

CREATE TRIGGER "Goods_BD" FOR "Goods" ACTIVE
BEFORE DELETE POSITION 0
AS
BEGIN
  update "Categories" set
    "GoodsCount" = (select count("Id") from "Goods" where
"IdCategory" = Old."IdCategory")
  where "Id" = Old."IdCategory";
END^
```

Давайте рассмотрим последовательность происходящих действий, например при добавлении нового наименования товара для какой-то категории.

В GoodsDataSet формируется значение ключевого поля "Id" при помощи генератора.

Значения полей заполняются пользователем.

После нажатия пользователем клавиши Enter GoodsData подготавливает запрос из свойства InsertSQL к выполнению и заполняет значения параметров соответствующими значениями полей.

Параметр :"IdCategory" заполняется текущим значением поля "Id" из CategoriesDataSet.

Запрос выполняется и выполняется триггер "Goods_AI".

Транзакция GoodsWriteTransaction автоматически подтверждает сделанные изменения при помощи метода Commit.

Выполняется запрос из свойства RefreshSQL, который перечитывает значения полей вставленной в таблицу "Goods" записи, основываясь на известном значении ключевого поля "Id".

В этот момент значение поля "GoodsCount" уже изменилось, но это изменение никак не отражается в CategoriesGrid. Мы должны обновить текущую запись в CategoriesDataSet, чтобы увидеть актуальное значение всех полей. Это можно сделать при помощи явного вызова метода CategoriesDataSet.Refresh. Однако этот же метод нам надо будет вызывать и в случае удаления записи из GoodsDataSet. А если бы нам нужно было создать более сложный механизм взаимодействия между master и detail-таблицами, то, скорее всего, метод Refresh нужно было бы вызывать также и после многих других операций.

Вместо всего этого мы можем просто добавить ключ dcForceMasterRefresh. Этот ключ автоматически вызывает метод Refresh у master, если detail была изменена. Если вы запустите приложение, то сами увидите, что значение поля "GoodsCount" автоматически изменяется при добавлении и удалении записей в GoodsDataSet.

Специальные опции в компонентах FIBPlus

Опции и настройки TpFIBDatabase

TpFIBDatabase содержит ряд специальных свойств, которые управляют режимами работы компонента. Мы уже описывали основные вещи, связанные с подключением к базе данных, однако некоторые особенности TpFIBDatabase остались "за кадром".

Хочется начать с описания механизма поддержки Alias в FIBPlus. Те из вас, кто уже работал с Interbase через Borland Database Engine, наверняка знают про существование такого понятия, как Database Alias, т. е. псевдоним базы данных. С точки зрения BDE Database Alias – это не что иное, как ряд параметров подключения к базе данных, которые хранятся в системном реестре Windows. Программист, который разрабатывает пользовательскую программу, подключающуюся к базе данных, может не указывать реальные параметры подключения, а просто "сослаться" на существующий алиас. Таким образом, если потом, например, хотелось поменять путь к базе данных, то достаточно было изменить информацию в алиасе и перезапустить программу.

Аналогичный механизм реализован и в FIBPlus. Вы можете задать свойство TpFIBDatabase.**AliasName** и забыть про такие параметры, как путь к базе данных, имя пользователя, character set, SQL Role и SQL Dialect. При запуске программы, если значение свойства AliasName не пустое, TpFIBDatabase будет пытаться получить значения соответствующих свойств из системного реестра. Встает вопрос: а как записать туда первоначальные значения? Для этого существует два способа.

Первый заключается в том, что вы устанавливаете свойство SaveDBParams в True, после чего все необходимые свойства автоматически сохраняются в системном реестре, как только вы подключитесь к базе данных. Фактически это одноразовая операция: установили свойства и имя алиаса, установили SaveDBParams в True, подключились к базе данных – параметры алиаса сохранены. Отключились от базы данных, выставили SaveDBParams в False. Теперь при последующих подключениях значения для свойств будут браться из системного реестра.

Второй способ – использование утилиты AliasManager, которая написана специально для того, чтобы можно было настраивать значения алиасов на любом клиентском месте без сред Delphi и C++ Builder. Иными словами, вы можете использовать эту утилиту на тех компьютерах, на которые установите свою программу. Мы не будем приводить здесь описание утилиты, поскольку ее использование достаточно очевидно само по себе (рис. 2.29).

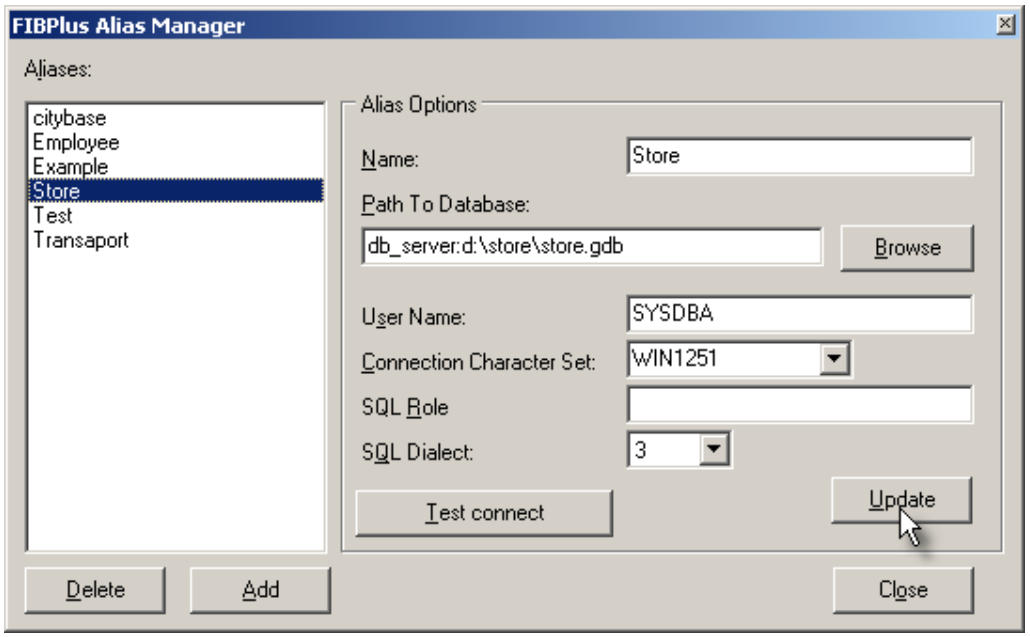


Рис. 2.29. Внешний вид Alias Manager

Достаточно удобным и полезным также является свойство **DesignDBOptions**. Если выставить ключ ddoIsDefaultDatabase в True, то у всех компонент (TpFIBDataSet, TpFIBQuery, TpFIBTransaction), которые вы потом положите на форму, автоматически будет заполняться свойство DefaultDatabase (или Database). Ключ ddoStoreConnected указывает – сохранять ли значение свойства TpFIBDatabase.Active при сохранении формы или проекта. Дело в том, что активное состояние базы данных зачастую бывает полезно в процессе разработки, когда мы используем визуальные средства FIBPlus: Database Editor, Transaction Editor, SQL Generator и различные свойства. Тем не менее, сохраняя активное

подключение (а значит, и автоматически восстанавливая его при загрузке программы!), мы можем получить ошибку в работе приложения, которое может попытаться открыть уже активное подключение. Чтобы избежать подобной накладки, вы можете выставить свойство ddoStoreConnected в False.

Свойство **SynchronizeTime** предназначено для правильного вычисления default-значений полей. Вы знаете, что в Interbase существуют специальные функции для работы с датами и временем: NOW и CURRENT_TIME (в Firebird). Эти функции часто используют при описании полей:

```
CREATE TABLE NODES_LIST (
  ID INTEGER NOT NULL,
  INSERTED TIME NOT NULL DEFAULT 'NOW'
);
```

При вставке записи с пустым значением поля INSERTED сервер автоматически подставит текущее время. Однако существует некоторая особенность при работе клиентской части программы. Как вы понимаете, чтобы работать с таблицей NODES_LIST, нам придется заполнять значения свойств SelectSQL, InsertSQL, DeleteSQL, UpdateSQL и RefreshSQL у некоторого компонента TpFIBDataSet. При вставке TpFIBDataSet проверяет свойства полей таблицы и пытается автоматически заполнить поля новой записи значениями по умолчанию. В нашем случае он должен присвоить полю INSERTED текущее время. Однако поскольку мы можем иметь дело с сетевой системой, то время на сервере может отличаться от времени на клиентской машине. В нашем гипотетическом примере это, возможно, и не играет никакой роли, однако могут быть случаи, когда необходимо подставлять точное "серверное" время. Для этого и используется свойство SynchronizeTime. Если оно выставлено в True, то при подключении к базе данных TpFIBDatabase выясняет "разницу во времени" между сервером и клиентским компьютером, чтобы впоследствии подставлять правильные значения.

Свойство **UpperOldNames** было введено в FIBPlus на этапе перехода с Interbase 5.x на Interbase 6.x. Как вы знаете, в 3-м SQL Dialect Interbase 6.x позволяет использовать "регистро-чувствительные" имена объектов. В старых версиях Interbase имена INSERTED и inserted являлись одинаковыми, поэтому программисты в своих программах не придерживались точного написания. В частности, можно было писать и следующее:

```
MainDS.FieldByName('COST').AsFloat :=
ainDS.FieldByName('cost').AsFloat * 0,3;
```

И это работало без ошибок. Однако если просто перенести базу на Interbase 6.x и запустить то же самое приложение, то оно начинало работать с ошибками, поскольку поле COST действительно существовало, а вот поле cost – уже нет. Фактически подобная миграция означала, что нужно было проверять все приложение, чтобы найти подобные подводные камни. FIBPlus позволяет обойтись без подобных мероприятий. Если выставить UpperOldNames в True, то любые идентификаторы, в которых явным образом не используются двойные кавычки, приводятся "внутри" к верхнему регистру, а значит, запись

```
MainDS.FieldByName('COST').AsFloat :=
ainDS.FieldByName('cost').AsFloat * 0,3;
```

снова начинает работать без проблем.

Опции и настройки TpFIBDataSet

TpFIBDataSet содержит гораздо больше настроек, которые регулируют его работу. Будет трудно, пожалуй, выбрать наиболее важные, поскольку все они предназначены для совсем разных задач, а поэтому начнем по порядку.

AllowedUpdateKinds. Свойство представляет собой множество из трех элементов: [okModify, ukInsert, ukDelete]. Достаточно очевидно, что, используя это свойство, мы можем разрешать и запрещать выполнение определенных модифицирующих операций TpFIBDataSet, не изменяя при этом свойства InsertSQL, DeleteSQL и UpdateSQL. То есть мы можем получить TpFIBDataSet, который будет считаться "живым", но не будет позволять, например, вставлять новые записи! Это, кстати, одна из стандартных проблем при работе с компонентом TDBGrid, который не позволяет ограничивать изменение данных, – он либо "совсем" редактируемый (а значит, пользователь может изменять поля, удалять записи и вставлять новые), либо "совсем" нередактируемый. При работе с FIBPlus мы можем регулировать это поведение без необходимости писать какие-то специальные обработчики событий. Есть и еще одна особенность в свойстве AllowedUpdateKinds. Как вы уже в курсе, Interbase позволяет регулировать права доступа к объектам базы данных. Таким образом, может возникнуть ситуация, когда конкретный пользователь, работающий с вашей программой, не имеет прав на удаление записей из конкретной таблицы. Писать для каждого пользователя отдельную программу, очевидно, слишком накладно. Однако TpFIBDataSet легко справляется с проблемой, поскольку если при попытке "скомпилировать" конкретный модифицирующий запрос компонент получает от сервера сообщение об ошибке из-за отсутствия прав, то TpFIBDataSet просто запрещает данное действие, убрав соответствующий элемент из свойства AllowedUpdateKinds. Дальнейшие попытки просто, таким образом, пресекаются, а пользователь работает с программой, даже не подозревая об ошибках или отсутствии прав.

Свойство **DefaultFormats** используется при заполнении свойства DisplayFormat у объектов TField при открытии запроса. Поэтому, если вы хотите автоматически задавать какой-то нужный вам формат представления, например, числовых полей, то вам достаточно указать его в DefaultFormats.NumericDisplayFormat. Правила заполнения масок можно прочитать в описании свойства DisplayFormat у стандартного класса TField.

Свойство **Options** тоже управляет режимами работы TpFIBDataSet, причем надо иметь в виду, что эти опции управляют поведением TpFIBDataSet уже после открытия запроса или в момент редактирования данных:

poTrimCharFields – среди разработчиков идут постоянные споры, должны ли компоненты для работы с Interbase "обрезать" конечные пробелы у строковых полей или нет. С одной стороны, дополнение пробелами – это описанная техническая особенность Interbase, а с другой стороны – это не всегда удобно для пользователя. Нет нужды спорить, если можно отрегулировать этот процесс в отдельно взятом приложении (и даже в отдельно взятом запросе). Включив

ключ PoTrimCharFields, вы можете быть уверены, что пробелов в конце строковых полей не будет.

poRefreshAfterPost позволяет вам отключить выполнение "обновляющего" запроса после редактирования записи. Как уже было сказано, после операции Post TpFIBDataSet автоматически выполняет запрос из свойства RefreshSQL, чтобы получить актуальные значения всех полей измененной записи, поскольку они могли быть изменены при помощи триггеров. Это дополнительный запрос, и если вы считаете, что в нем нет никакой нужды (например, вы точно уверены, что никакие триггеры не изменяют записи в таблице), то вы можете отключить этот режим.

При включенной опции *poRefreshDeletedRecord* если в результате выполнения RefreshSQL запрос ничего не вернул (в частности, это может произойти, если запись была удалена другим пользователем), то TpFIBDataSet также удалит ее из внутреннего буфера.

poStartTransaction отвечает за автоматический запуск транзакции перед открытием запроса, если транзакция еще не была активна. Если вы предпочитаете вызывать метод StartTransaction вручную, то можете отключить данную опцию.

poAutoFormatFields разрешает или запрещает автоматическое форматирование полей на основе форматов, заданных в свойстве DefaultFormats.

poProtectEdit руководит режимом "защищенного" редактирования. Этот режим иногда полезен, если вы хотите избежать коллизий при редактировании одной и той же записи разными пользователями. Если активизировать режим, то, как только вы начинаете редактировать запись (но до вызова метода Post), на сервер отправляется так называемый "холостой" update вида:

```
UPDATE TABLE
SET FIELD1 = FIELD1,
FIELD2= FIELD2, …
```

То есть запрос, который формально является редактированием записи, а фактически не изменяет значения полей. Тем не менее сервер считает, что изменения произошли, а значит, пользователь, который первым начал редактирование, имеет преимущество перед всеми последующими. Фактически это эмуляция режима блокировок на уровне записей. Проблема, однако, состоит в том, что в Interbase не существует возможности отменить UPDATE, не отменив транзакции целиком. Поэтому даже если пользователь начал редактирование записи, а потом отказался и метод Post так и не был вызван, то запись все равно будет считаться измененной до конца транзакции. Кроме того, данный режим практически совершенно бесполезен, если вы используете режим работы в контексте двух транзакций, как это было описано выше, поскольку "блокировка" действует только до конца изменяющей транзакции.

Опция *poKeepSorting* нужна, если вы используете методы локальной сортировки DoSort и DoSortEx. Эти методы выполняют сортировку записей внутри локального буфера (без запроса на сервер). При изменении значений полей возможна ситуация, что измененная запись должна быть помещена в другое место (если изменилось, например, значение поля, по которому записи отсортированы). При активной опции poKeepSorting, после изменения записи, весь буфер автоматически сортируется и запись "встает" на нужное место.

poPersistentSorting – при активной опции, если вы вызывали методы DoSort или DoSortEx, то после переоткрытия запроса результат будет автоматически отсортирован вызовом DoSort с теми же параметрами.

Опция *poAllowChangeSQLs* блокирует или разрешает изменение свойств InsertSQL, DeleteSQL, UpdateSQL и RefreshSQL при открытом запросе.

Свойство **PrepareOptions**, в отличие от Options, позволяет задать некоторые особенности поведения TpFIBDataSet до открытия запроса:

pfSetRequiredFields – если опция активна, то для полей, которые в базе данных были описаны как NOT NULL, свойство TField.Required будет выставлено в True. Если пользователь попробует не задать значение для таких полей, например, в TDBGrid, то на экран будет выведено стандартное предупреждение.

При включенной опции *pfSetReadOnlyFields* для полей, которые описаны в базе как вычислимые (а значит, не изменяемые пользователем), свойство TField.ReadOnly будет выставлено в True. Это позволит избежать попыток изменения значений таких полей пользователем.

pfImportDefaultValues – данная опция включает и выключает режим, при котором TpFIBDataSet будет автоматически заполнять поля их значениями по умолчанию при вставке новой записи.

Опция *psUseBooleanFields* включает или выключает режим эмуляции Boolean-полей. Вы знаете, что Interbase не поддерживает специальный тип для логических значений и разработчики вынуждены так или иначе заменять его своим пользовательским типом. Однако даже пользовательские типы не дают равноценной замены Boolean-типу, поскольку визуальные компоненты все равно видят такие поля либо как числовые, либо как строковые в зависимости от описания. Таким образом, пропадает возможность использования каких-то специальных возможностей отображения логических полей визуальными компонентами (например, многие сторонние компоненты наподобие TDBGrid отображают логические поля галочками). FIBPlus позволяет эмулировать Boolean-поля гораздо более эффективно. Необходимо описать в базе данных домен вида:

```
CREATE DOMAIN T_BOOLEAN
AS INTEGER DEFAULT 0
NOT NULL CHECK (VALUE IN (0,1))
```

Поле должно быть целочисленным и содержать в названии слово "boolean". После этого все поля, которые вы создадите с использованием данного домена, будут "распознаваться" в FIBPlus как Boolean-поля, т. е. для них будут создаваться экземпляры класса TFIBBooleanField, который является прямым потомком стандартного TBooleanField, значит, любой визуальный компонент будет работать с такими полями как с настоящими Boolean!

psSQLINT64ToBCD – опция заставляет TpFIBDataSet создавать экземпляры TBCDField для полей INT64 с любым масштабом (в SQL Dialect 3), что гарантирует точность вычислений для любых полей INT64.

psApplyRepositary – опция включает или выключает режим использования FIBPlus Repository. Эта особенность FIBPlus требует отдельного разговора, и мы вернемся к ее рассмотрению позже в отдельном параграфе.

psGetOrderInfo – при включенном режиме, FIBPlus анализирует запрос перед выполнением и автоматически заполняет структуру SortFields данными о полях,

по которым происходит сортировка в запросе. Впоследствии это может быть использовано в методах локальной сортировки DoSort и DoSortEx.

Опция *psAskRecordCount* предназначена для получения информации о количестве записей, которые должен вернуть запрос, до выполнения, собственно, запроса. Дело в том, что фактически, значение свойства TpFIBDataSet.RecordCount показывает лишь количество записей, которые мы уже получили с сервера. Это число может отличаться от полного количества записей, поскольку если мы не вызывали метод FetchAll, то сервер "отдает" нам записи по мере необходимости (при вызове метода Next). Иногда бывает полезно узнать точное количество заранее. В этом случае нужно включить опцию psAskRecordCount. Однако необходимо учитывать, что данная опция имеет ряд ограничений. Поскольку единственным способом узнать количество записей в запросе заранее является выполнение другого запроса вида SELECT COUNT (*) с теми же условиями, что и в основном запросе, то для некоторых запросов мы выполнить такую операцию не можем. Например, если мы имеем запрос

```
SELECT ID FROM TABLE1
UNION
SELECT ID FROM TABLE2
```

то запрос SELECT COUNT(*) … попросту теряет смысл, поскольку результат будет ложным по определению. В общем случае опция psAskRecordCount не работает с запросами, которые содержат команды UNION, ORDER и DISTINCT.

Использование FIBPlus совместно с генератором отчетов FastReport

Фактически компоненты FIBPlus совместимы с любыми генераторами отчетов, которые работают с потомками стандартного класса TDataSet. Таким образом, вы сможете использовать FIBPlus вместе с QuickReport, ReportBuilder, FastReport и даже со специализированными генераторами отчетов, такими, как, например, Afalina XLReport.

Все эти генераторы будут получать данные из TpFIBDataSet напрямую или через компонент TDataSource. Мы не будем останавливаться на рассмотрении всех особенностей всех генераторов, а просто продемонстрируем как быстро и просто можно получить отчет при помощи FastReport от FastReport Software. Отметим, что, как и FIBPlus, этот генератор отчетов можно использовать не только при разработке приложений в Borland® Delphi® для Microsoft® Windows®, но и в Borland® Kylix® для Linux®. Для этого предназначена специальная CLX-версия пакета.

Простой отчет

Не претендуя на демонстрацию всех возможностей FastReport, поскольку этот пакет предоставляет очень мощные и гибкие возможности для построения отчетов, мы создадим отчет, который распечатает наш price-лист.

Идеология FastReport позволяет включать "внутрь" отчетов совершенно независимые источники данных (и даже отдельные независимые подключения к базе данных), чтобы получить, таким образом, отчеты, которые можно было бы создавать отдельно от приложения (например, если первоначальная постановка задачи не предусматривала каких-то специальных отчетов). Это потребует установки дополнительных компонентов для "привязки" к конкретному источнику данных. Вместе со стандартным пакетом FastReport поставляются, в частности, компоненты для использования FasrtReport с IBX, ADO, BDE и FIBPlus. Автором компонентов FastReport для FIBPlus является Виталий Бармин (barmin@udm.ru).

Итак, если вы хотите использовать и дополнительные возможности FastReport вместе с FIBPlus (этот вопрос будет рассмотрен в разделе "Генератор отчетов FastReport. Создание отчетов в run-time"), то вам необходимо установить соответствующий пакет (FrFib3.dpk, FrFib4.dpk, FrFib5.dpk или FrFib6.dpk, в зависимости от версии Delphi) входящий в поставку FastReport.

> Вы также можете всегда найти эти компоненты Виталия Бармина на сайте FIBPlus: http://www.fibplus.net/.
>
> Напомним, что сам FastReport вы всегда сможете найти на сайте http://www.fastreport.ru.

После установки вы увидите в палитре на странице FastReport дополнительный компонент: TfrFIBComponents. Теперь можно перейти к созданию отчета.

Поместим на нашу форму следующие компоненты: frReport: TfrReport; frFIBComponents: TfrFIBComponents и frDBDataSet1: TfrDBDataSet (рис. 2.30).

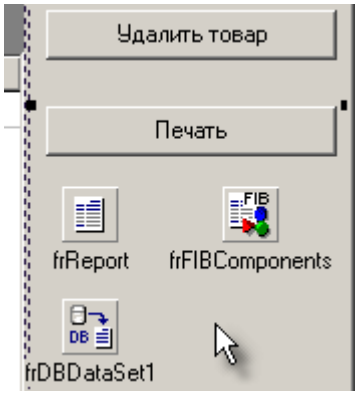


Рис. 2.30. Компоненты FastReport

Откроем дизайнер отчетов, дважды нажав на frReport (рис. 2.31).

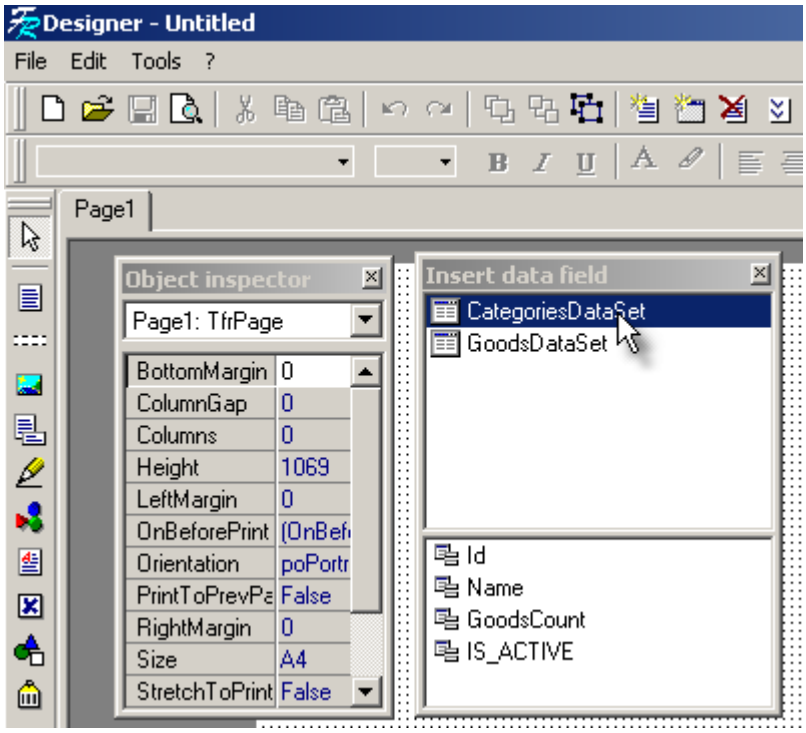


Рис. 2.31. Внешний вид дизайнера отчетов

Как видно из рисунка, оба компонента TpFIBDataSet уже доступны нам в дизайнере. Для начала рассмотрим самый простой вариант отчета, который распечатает нам список категорий товаров в прайс-листе (рис. 2.32).

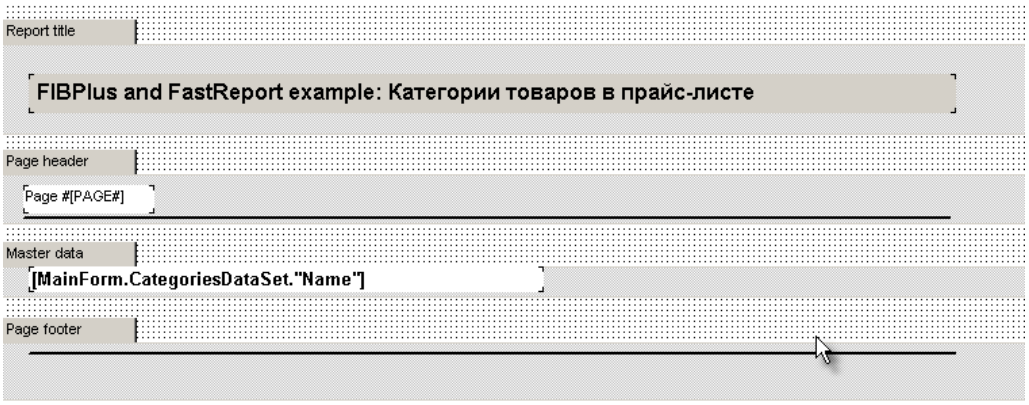


Рис. 2.32. Шаблон отчета о категориях товаров

Для этого мы поместим на страницу четыре полосы (band): Report Title, Page Header, Master Data и Page Footer. Объекты, помещенные на Report Title, будут распечатаны только один раз в самом начале отчета. Объекты, размещенные на Page Header, будут выводиться на печать в начале каждой новой страницы. В нашем случае мы просто добавили туда переменную PAGE#, которая будет выводить номер страницы. Master Data нужен нам для вывода данных из компонента CategoriesDataSet. Эта полоса (band) будет выведена на печать столько раз, сколько записей в нашем наборе данных. И наконец, Page Footer будет выводиться в конце каждой страницы отчета.

Укажем FastReport, откуда необходимо брать данные для отчета. Для этого зададим свойство DataSource у компонента frDBDataSet1 (рис. 2.33).

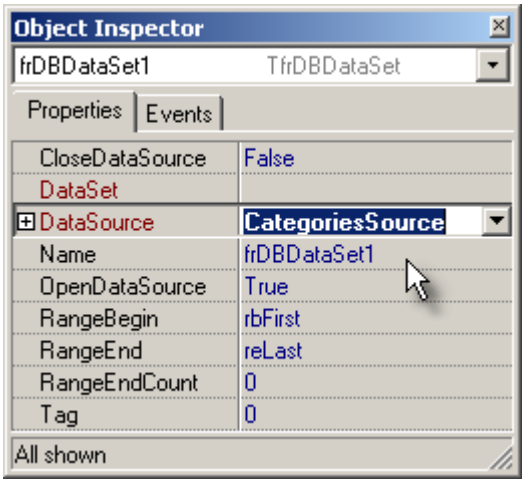


Рис. 2.33. Свойства TfrDBDataSet

Теперь необходимо задать источник данных для Master Data Band в самом отчете. Нажмите дважды на Master Data в дизайнере отчета и укажите источник данных в появившемся диалоге (рис. 2.34).

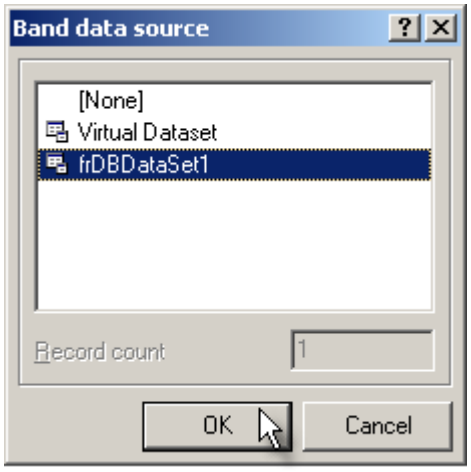


Рис. 2.34. Источник данных для Master Data

Остается обратить внимание на объект TfrMemoView, в котором и будут выводиться названия категорий в отчете (рис. 2.35).

Рис. 2.35. Объект шаблона отчета TfrMemoView, необходимый для печати значений поля таблицы

Это выражение вы можете ввести вручную, а можете использовать специальный диалог редактирования объекта и нажать кнопку Insert data field, изображенную на рис. 2.36.

На этом этапе фактически отчет готов. Вы можете нажать кнопку Preview и увидеть уже готовый список категорий, который можно распечатать. Тем не менее добавим в наше приложение кнопку, которая будет вызывать отчет по названиям категорий товаров.

Необходимо написать очень простую обработку нажатия на эту кнопку (предположим, что мы сохранили шаблон для отчета в файле 'price_categories.frf):

```
procedure TMainForm.PrintBClick(Sender: TObject);
begin
  frReport.LoadFromFile('price_categories.frf');
  frReport.PrepareReport;
  frReport.ShowPreparedReport;
end;
```

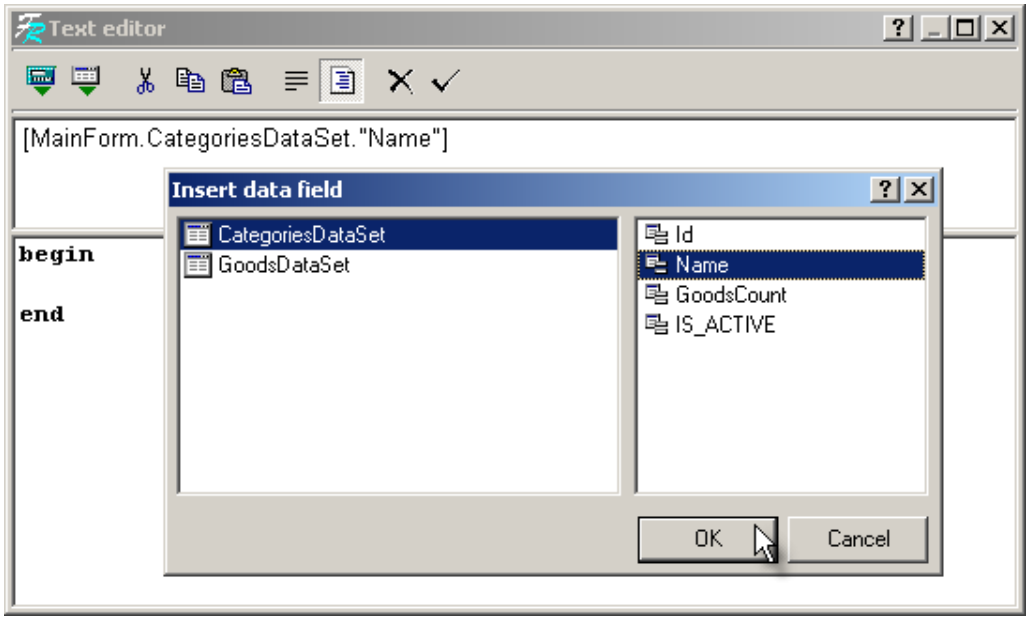


Рис. 2.36. Внешний вид диалога редактирования объекта TfrMemoView

После запуска приложения и нажатия на кнопку Печать мы увидим следующий отчет (рис. 2.37).

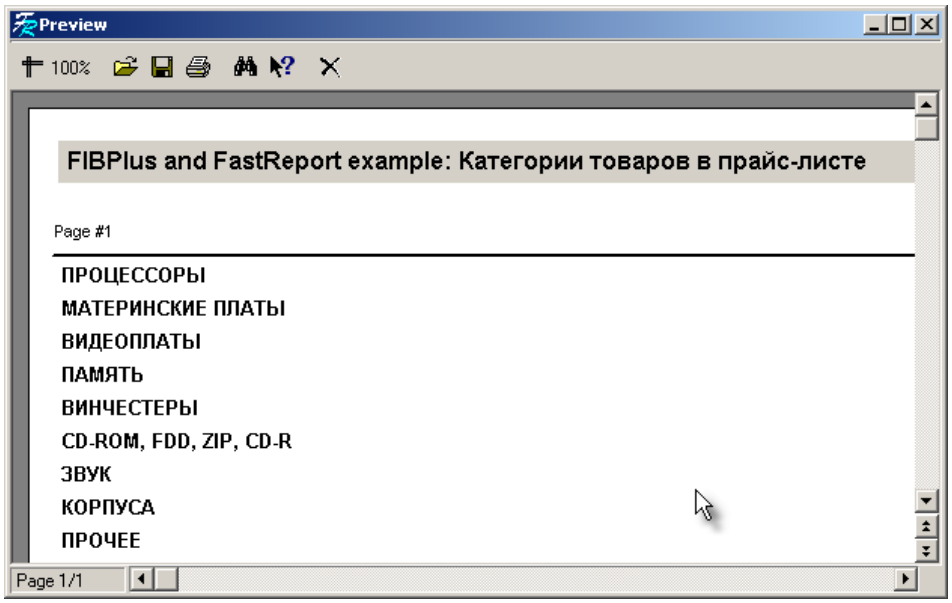


Рис. 2.37. Вид отчета о категориях товаров в прайс-листе

Отчеты вида master-detail

Теперь мы можем приступить к построению отчета более сложной структуры, хотя фактически это реализуется так же просто, как и предыдущий вид отчета. Сначала добавим на нашу форму еще один компонент frDBDataSet2: TfrDBDataSet – и зададим его свойство DataSource равным GoodsSource. Теперь необходимо добавить две полосы (bands): Detail Header (заголовки детализованных данных) и Detail Data (сами данные). На Detail Data мы разместим два объекта типа TfrMemoView. В первом мы будем выводить названия товаров, а во втором – цену каждого товара (рис. 2.38):

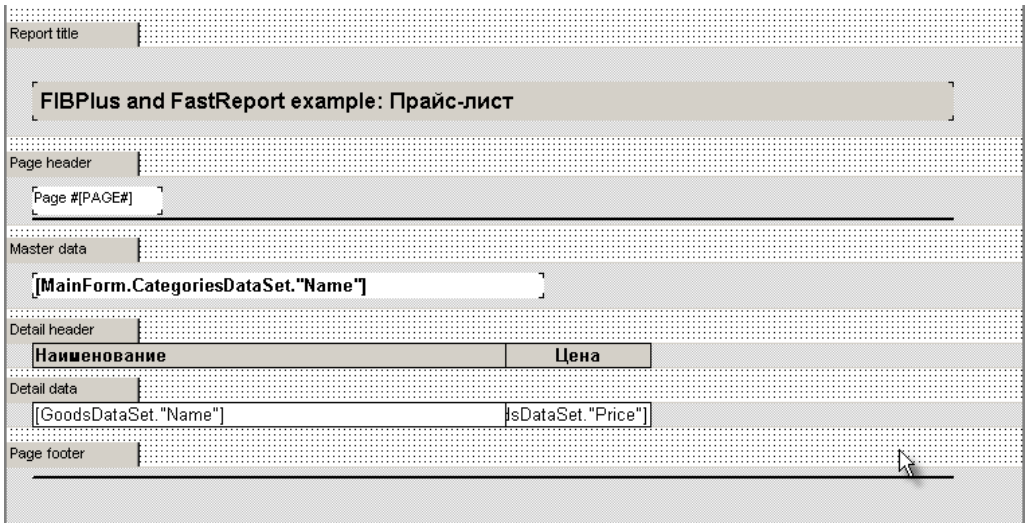


Рис. 2.38. Шаблон отчета для связи мастер-деталь

Очевидно, что мы должны указать источником данных для Detail Data компонент frDBDataSet2. Отчет готов. Сохраним его в файл 'price_list.frf' и изменим немного обработчик клавиши "Печать":

```
procedure TMainForm.PrintBClick(Sender: TObject);
begin
  GoodsDataSet.DetailConditions := GoodsDa-
taSet.DetailConditions - [dcWaitEndMasterScroll];
  frReport.LoadFromFile('price.frf');
  frReport.PrepareReport;
  frReport.ShowPreparedReport;
  GoodsDataSet.DetailConditions := GoodsDa-
taSet.DetailConditions + [dcWaitEndMasterScroll];
end;
```

> Важный момент касается изменения DetailConditions. Как уже было сказано, ключ dcWaitEndMasterScroll позволяет избегать лишних запросов при навигации по master-запросу. Однако в случае с распечаткой полного отчета это может привести к тому, что для каждой категории товара мы будем иметь одни и те же наименования, поскольку GoodsDataSet не будет переоткрывать запрос, пока FastReport будет получать данные из CategoriesDataSet. Чтобы избежать такой ошибки, мы отключаем оптимизацию master-detail на период подготовки отчета и включаем ее вновь после того, как отчет уже готов.

Теперь при запуске приложения мы можем легко распечатать весь наш прайс-лист (рис. 2.39).

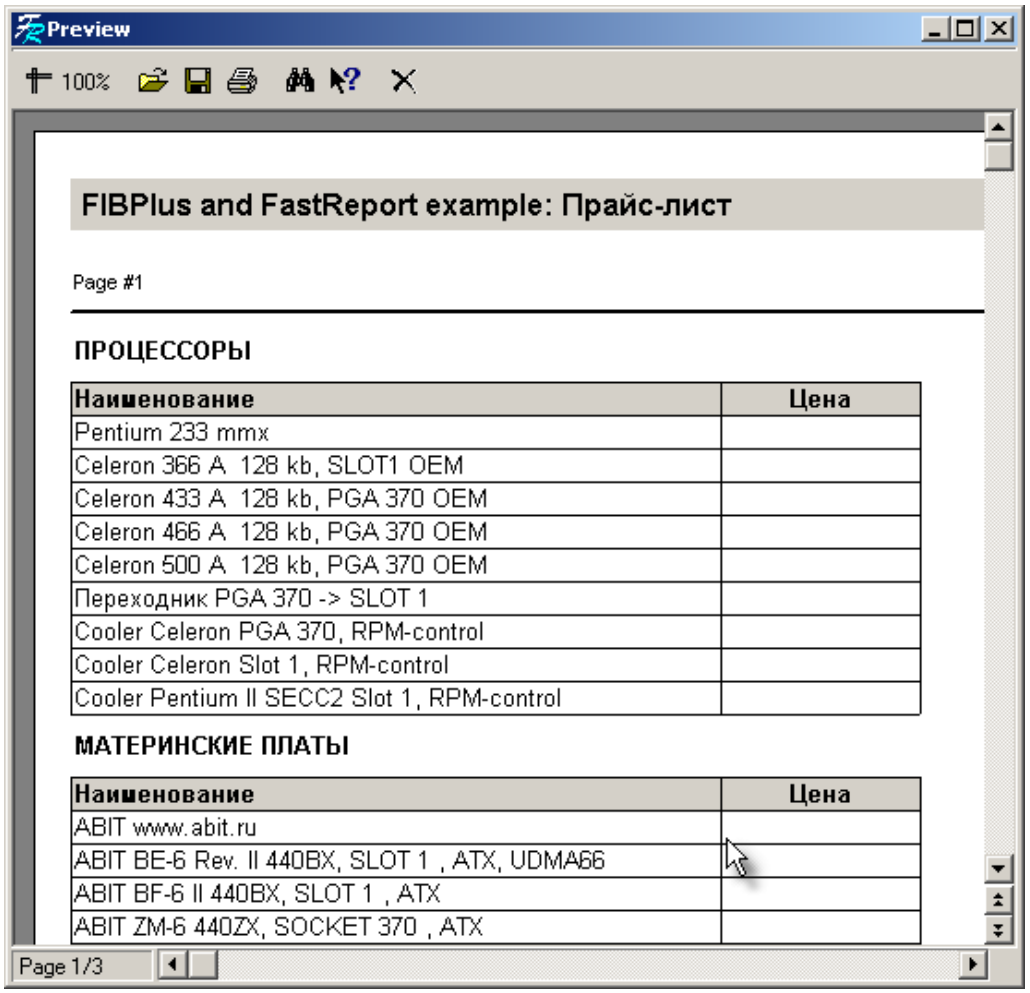


Рис. 2.39. Внешний вид отчета вида мастер-деталь

При желании мы можем выделить цветом или курсивом "горячие" (к примеру, самые дешевые) позиции прайса. Для этого в дизайнере выберем TfrMemoView, затем воспользуемся ObjectInspector и установим свойство Highlight. Для этого в поле "Condition" появившегося окна введем условие Value<0 и выберем способ выделения. В нашем примере все позиции прайса с ценой менее 1000 будут выделены ярко-зеленым цветом (рис. 2.40).

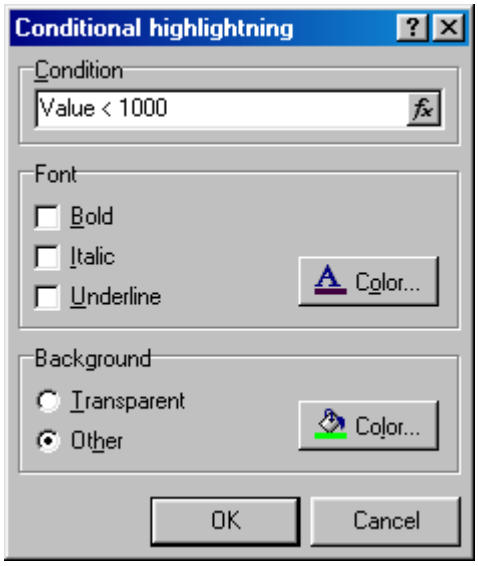


Рис. 2.40. Диалог параметров условного форматирования

Создание отчетов в run-time

Одной из самых замечательных особенностей FastReport является возможность создавать отчеты прямо в run-time, т. е. во время работы приложения. Это позволяет добавлять в ваше приложение отчеты, которые не были предусмотрены заранее, и использовать в этих отчетах как существующие запросы, так и совершенно новые. Более того, фактически вы можете создавать в таких отчетах даже самостоятельные подключения к базам данных не меняя код вашей программы!

Все это возможно благодаря архитектуре FastReport и компонентам, написанным Виталием Барминым, о которых мы уже упоминали выше.

Для этого добавим в наше приложение компоненты TfrDesigner, TfrDialogControls (как видно из названия, этот компонент предназначен для построения и использования самостоятельных диалоговых окон в составе отчета), а также кнопку для вызова диалога редактирования шаблона отчета (рис. 2.41).

Обработчик для этой кнопки будет исключительно простым:

```
procedure TMainForm.DesignPClick(Sender: TObject);
begin
  frReport.DesignReport;
end;
```

Теперь при запуске приложения мы можем нажать на кнопку "Дизайн отчетов" и создать отчет прямо в run-time, сохранить его в отдельном файле и потом использовать в приложении. Все эти шаги, в сущности, являются тем же самым, что мы уже делали в предыдущих разделах. Теперь же мы хотим сосредоточиться на дополнительных возможностях FastReport, а именно на создании независимых отчетов.

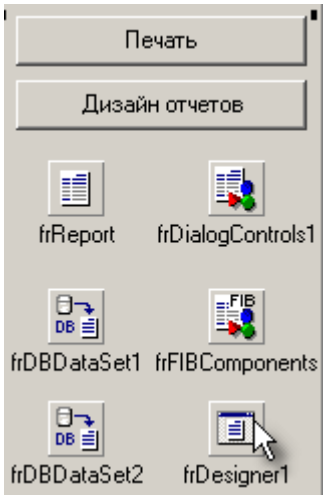


Рис. 2.41. Компоненты для встраивания дизайнера отчетов в программу

Если вы обратите внимание, то шаблон для отчета по умолчанию имеет одну страницу, на которой мы и располагаем полосы для печати. Добавим к отчету специальную страницу – диалоговую, на которой расположим компоненты доступа к данным (рис. 2.42).

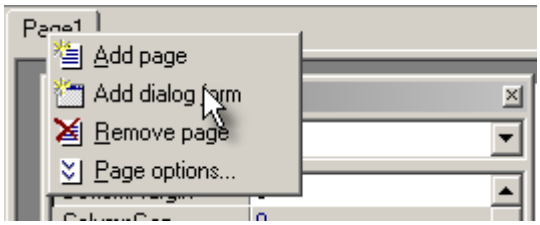


Рис. 2.42. Добавление диалоговой страницы к шаблону отчета

После добавления диалога и перехода на страницу 2 нашего шаблона на панели инструментов с объектами, доступными для отчета, вы увидите ряд визуальных компонентов, которые можно располагать на диалоге (рис. 2.43).

Рис. 2.43. Компоненты для доступа к базе данных из шаблона отчета

Положим на диалог компонент Query: TfrFIBQuery, укажем ему существующее подключение к базе данных в свойстве Database и напишем следующий запрос в свойстве SQL:

```
SELECT "Categories"."Name", "Categories"."GoodsCount"
FROM "Categories"
WHERE "Categories"."GoodsCount" > 0
```

Очевидно, что мы хотим вывести на печать только те категории товаров, количество наименований по которым больше нуля. Остается только создать сам шаблон печати, привязав соответствующие полосы (bands) к Query как к источнику данных (рис. 2.44).

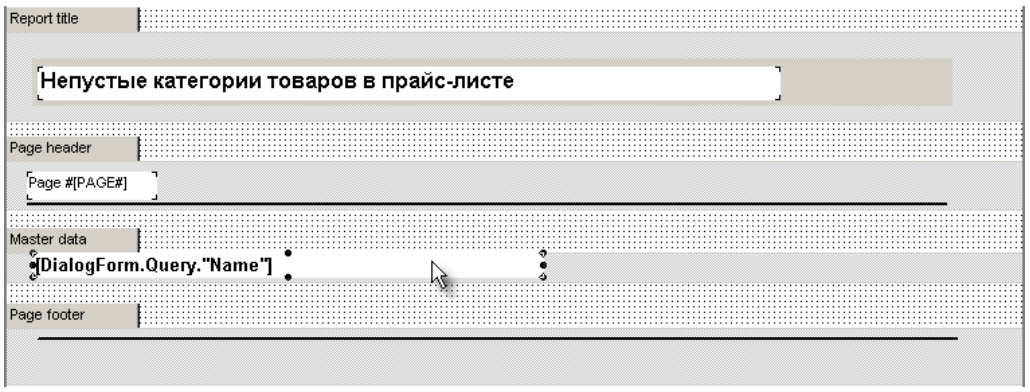


Рис. 2.44. Шаблон встроенного отчета

Укажем источник данных для Master Data (рис. 2.45).

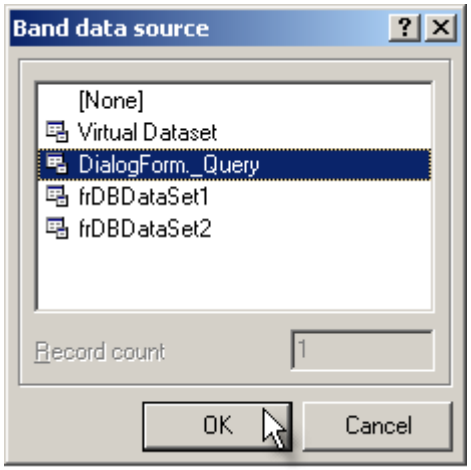


Рис. 2.45. Подключение полосы (band) к встроенному в шаблон запросу

И сохраним наш новый отчет, например, под именем "price1.frf". Теперь можно несколько изменить реакцию нашего приложения на нажатие кнопки печати. Положим на форму компонент OpenDialog: TopenDialog – и напишем следующий обработчик нажатия на кнопку "Печать":

```
procedure TMainForm.PrintBClick(Sender: TObject);
begin
  if not OpenDialog.Execute then exit;
  GoodsDataSet.DetailConditions := GoodsDa-
taSet.DetailConditions - [dcWaitEndMasterScroll];
```

```
  frReport.LoadFromFile(OpenDialog.FileName);
  frReport.PrepareReport;
  frReport.ShowPreparedReport;
  GoodsDataSet.DetailConditions := GoodsDa-
taSet.DetailConditions + [dcWaitEndMasterScroll];
end;
```

Таким образом, когда пользователь нажмет на кнопку "Печать", он сможет выбрать любой шаблон отчета и вывести его на печать. Учитывая, что при помощи дизайнера отчетов тот же самый пользователь (или вы, как разработчик) может подготовить неограниченное количество самых разных отчетов без необходимости изменения основного кода приложения, становится очевидным преимущество использования FIBPlus совместно с FastReport. Конечно, возможность создавать отчеты в run-time доступна в FastReport и для других пакетов работы с данными, в частности с IBX, однако FIBPlus делает эту возможность значительно более гибкой за счет использования макросов. Более подробно механизм макросов мы рассмотрим чуть позже, а сейчас лишь продемонстрируем, как, используя макросы, мы можем создавать более функциональные диалоговые отчеты.

Скопируем "price1.frf" в "price2.frf", откроем "price2.frf" для редактирования в дизайнере отчетов и перейдем на диалоговую страницу. Там мы поместим дополнительные компоненты: ComboBox1, Edit1, Label1 и Button1 (рис. 2.46).

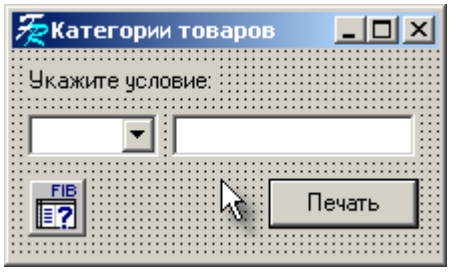


Рис. 2.46. Внешний вид диалога, встроенного в шаблон отчета

Изменим запрос в Query.SQL:

```
SELECT "Categories"."Name", "Categories"."GoodsCount"
FROM "Categories"
WHERE "Categories"."GoodsCount" @COND
```

COND в данном случае является макросом FIBPlus, и вместо него мы можем поставить любой текст. Удобство же заключается прежде всего в том, что макросы в FIBPlus рассматриваются как полноценные параметры, а значит, FastReport тоже будет считать, что в данном случае имеет дело с параметром. В данном примере мы можем позволить пользователю самому указать условия для выборки категорий прайс-листа. Заполним ComboBox1.Items символами сравнения:

```
>
<
>=
<=
<>
=
```

Теперь нам достаточно только указать FastReport, какое значение необходимо "подставить" вместо нашего макроса-параметра. Для этого надо изменить свойство Params у компонента Query (рис. 2.47).

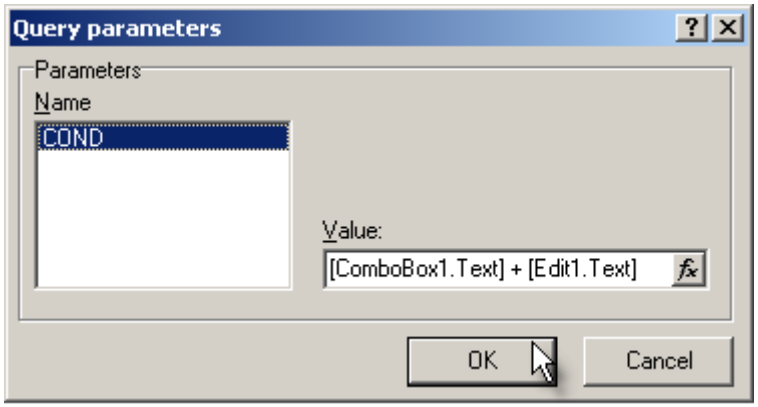


Рис. 2.47. Выражение для получения значение макроса COND

Параметр у нас один: COND. Его значение мы должны сформировать из двух частей: знака сравнения и значения, с которым будет сравниваться поле "GoodsCount". Что и было сделано, как видно из рисунка выше. Теперь при запуске отчета мы увидим диалог, в котором необходимо указать условие для выборки (рис. 2.48).

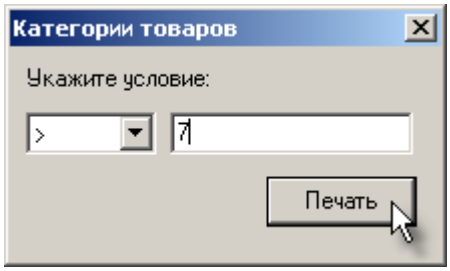


Рис. 2.48. Полученный диалог во время получения отчета

А на рис. 2.49 изображен получившийся отчет о категориях товаров, в которых более семи наименований товаров.

Теперь мы можем определить реакцию на щелчок по любой категории товаров в подобном отчете таким образом, чтобы по нему выводился полный отчет о товарах из этой категории. Для этого у всех объектов FastReport есть свойство Tag: String. В окне preview пользователь может нажать на объект и при этом выполнится какое-либо действие (например, построится другой отчет). Цель свойства – облегчить распознавание нажатого объекта. Например, объект может содержать текст "12.25р", по которому идентификация невозможна. Но в свойство Tag можно поместить более развернутую информацию, например значение ключа таблицы, из которой был получен текст "12.25р", номер строки или колонки в отчете и пр. При этом необходимо использовать событие TfrReport.OnObjectClick.

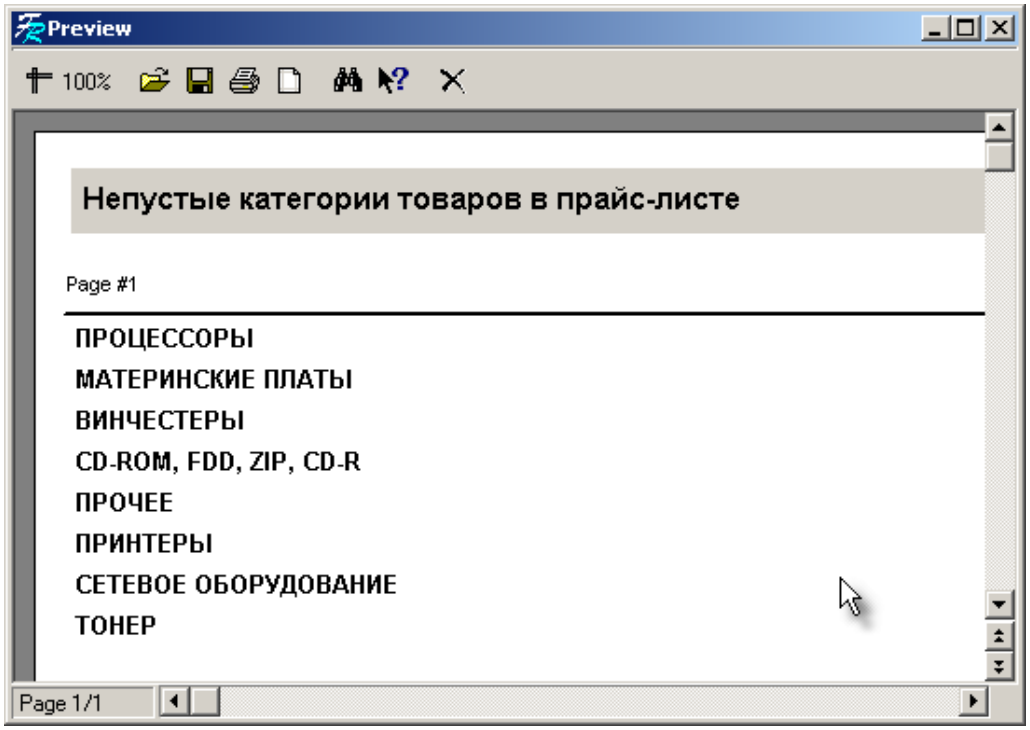


Рис. 2.49. Внешний вид получившегося динамического отчета

Логично использовать для вывода детализованного отчета тот же самый объект frReport, который уже лежит у нас на диалоговой форме приложения. Однако саму форму этого отчета надо будет разработать отдельно и сохранить на диске. Форма отчета будет выглядеть так же, как и в нашем отчете master-detail (рис. 2.50).

Кроме того, нам будет необходимо наложить фильтр на категории товаров:

```
SELECT "Categories"."Name", "Categories"."GoodsCount"
FROM "Categories"
WHERE "Categories"." Name " @NAME
```

Теперь укажем FastReport, какое значение следует подставить вместо макроса "@NAME" в компоненте Query. Здесь нам придется немного "схитрить" и присвоить этому макросу значение переменной FastReport. Назовем эту переменную Var1.

А самой переменной мы присвоим значение в коде программы.

Сохраним эту форму отчета из дизайнера на диске в текущем каталоге под именем "detailed.frf".

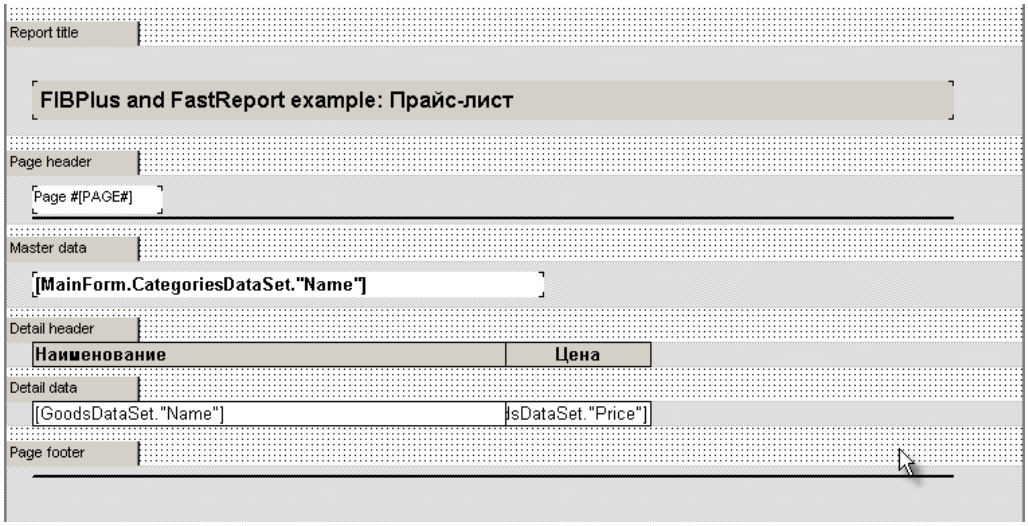


Рис. 2.50. Шаблон дополнительного отчета

Определим в коде программы для TfrReport событие OnObjectClick следующим образом:

```
procedure Tform1.frReportObjectClick(View: TfrView);
var str: string;
begin
str := View.Memo.Text;
frReport.LoadFromFile('detailed.frf')
frReport1.Dictionary.Variables['Var1'] := str;
if frReport.PrepareReport then  begin
frReport.OnObjectClick := nil; { чтобы при щелчке на детализо-
ванном отчете не показывался детализованный отчет }
frReport.ShowPreparedReport;
frReport.OnObjectClick := frReportObjectClick; { возвращаем со-
бытие для вывода детализованного отчета }
frReport.ShowReport;
end;
end;
```

> Наиболее полную документацию, описание всех возможностей, а также последние версии FastReport можно найти на официальном Web-сайте http://www.fastreport.ru.

Использование специальных инструментов в design-time: FIBPlus Tools

Кроме компонентов библиотека FIBPlus также включает ряд дополнительных инструментов – FIBPlus Tools, которые расширяют возможности среды разработки специально для более удобного и эффективного использования FIBPlus.

Установка FIBPlus Tools

FIBPlus Tools – это эксперты для Delphi и C++ Builder, поставляющиеся в готовом, скомпилированном виде, поэтому для их установки в среде необходимо установить соответствующий пакет.

> На момент создания книги опубликованы FIBPlus Tools для Borland Delphi 3–7 и Borland C++ Builder 5–6. Если вы используете другие версии продуктов Borland, мы рекомендуем вам проверить более новые версии FIBPlus Tools на сайте http://www.fibplus.net/.

Рассмотрим установку в среде Delphi (рис. 2.51). Необходимо выбрать пункт основного меню Components -> Install Packages.

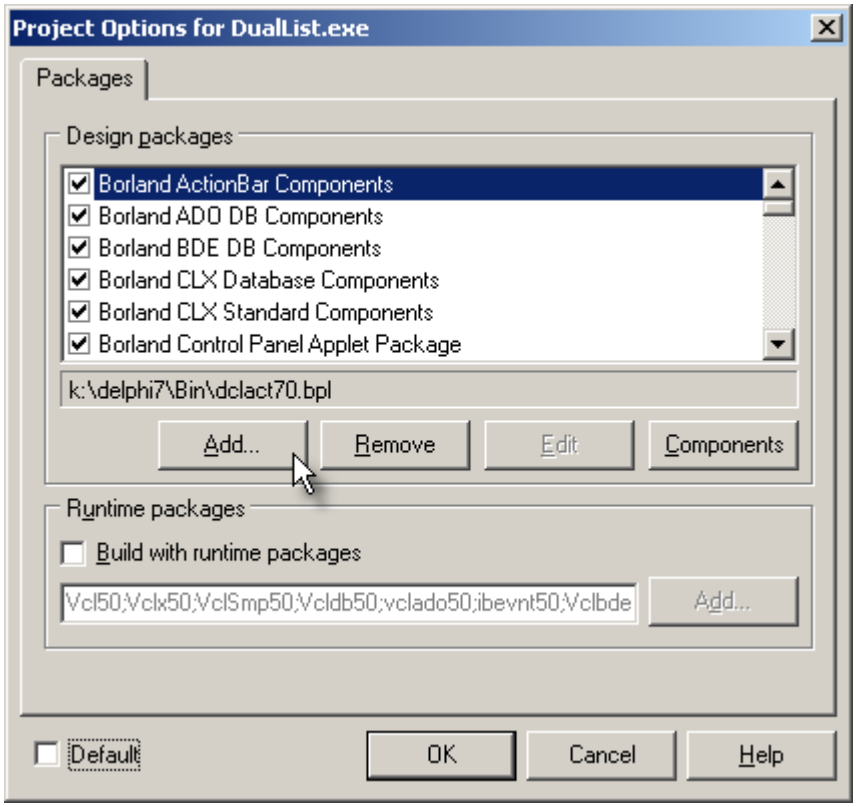


Рис. 2.51. Установка FIBPlusTools в среду Delphi

Нажмите на кнопку Add и найдите соответствующий вашей версии Delphi пакет (табл. 2.1).

Таблица 2.1. FIBPlusTools: Поддерживаемые версии Delphi/C++ Builder

Версия Delphi/C++ Builder	Название пакета FIBPlus Tools
Delphi 3	pFIBPlusTools3.dpl
Delphi 4	pFIBPlusTools4.bpl
Delphi 5	pFIBPlusTools5.bpl
Delphi 6	pFIBPlusTools6.bpl
Delphi 7	PFIBPlusTools7.bpl
C++ Builder 5	pFIBPlusTools_CB5.bpl
C++ Builder 6	pFIBPlusTools_CB6.bpl

После установки вы обнаружите пункт FIBPlus в основном меню Delphi (рис. 2.52).

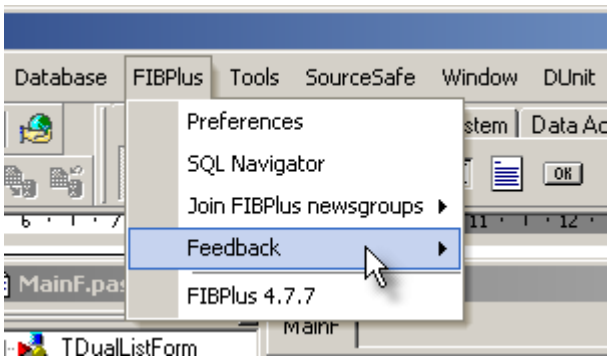


Рис. 2.52. Меню FIBPlus Tools

Последний пункт в меню FIBPlus фактически ничего не делает, но показывает номер установленной версии.

Preferences

Пункт Preferences позволяет настроить параметры основных компонентов по умолчанию (рис. 2.53).

На первой странице диалога вы можете задать значения по умолчанию для свойств Options, PrepareOptions и DetailsConditions для всех компонентов класса TpFIBDataSet. Вы можете задать определенные ключи для этих свойств. Например, если вы включите флажок SetRequiredFields то, когда вы положите новый компонент TpFIBDataSet на вашу форму, ее свойство PrepareOptions будет содержать ключ pfSetRequiredFields. Наиболее важным является тот факт, что умолчания, заданные в FIBPlus Tools Preferences, действуют во всех приложениях, которые вы будете создавать. Однако необходимо иметь в виду, что это

только первоначальные умолчания. То есть если после помещения компонента на форму вы измените свойства, то это никак не коснется Preferences. Также изменение Preferences не коснется тех компонент значения свойств которых уже были заданы.

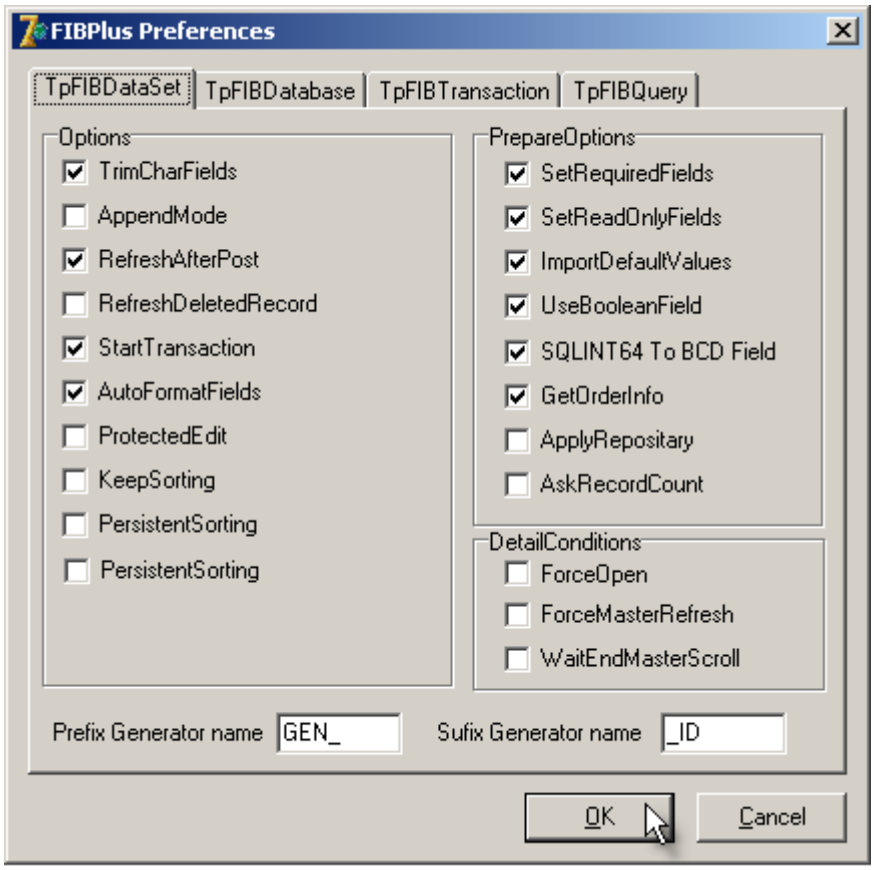


Рис. 2.53. Диалог настройки опций TpFIBDataSet в design-time.

Обратите внимание на поля "Prefix Generator name" и "Suffix Generator name". Задав их значения, вы сможете регулировать формирование имен для названий генераторов в свойстве AutoUpdateOptions у TpFIBDataSet. Имя генератора в AutoUpdateOptions генерируется из названия таблицы (UpdateTable), префикса и суффикса.

Следующие страницы диалога позволяют настраивать ключевые свойства компонентов TpFIBDataBase, TpFIBTransaction и TpFIBQuery.

В частности, если вы всегда работаете с новыми версиями InterBase, т. е. С версиями 6 и более (а также Firebird), то мы рекомендуем вам задать значение SQL Dialect на закладке TpFIBDatabase равным 3, чтобы каждый раз не переключать это свойство вручную.

SQL Navigator

Это наиболее интересная часть FIBPlus Tools, не имеющая аналогов в других продуктах. Фактически это инструмент централизованной обработки SQL в рамках целого приложения (рис. 2.54):

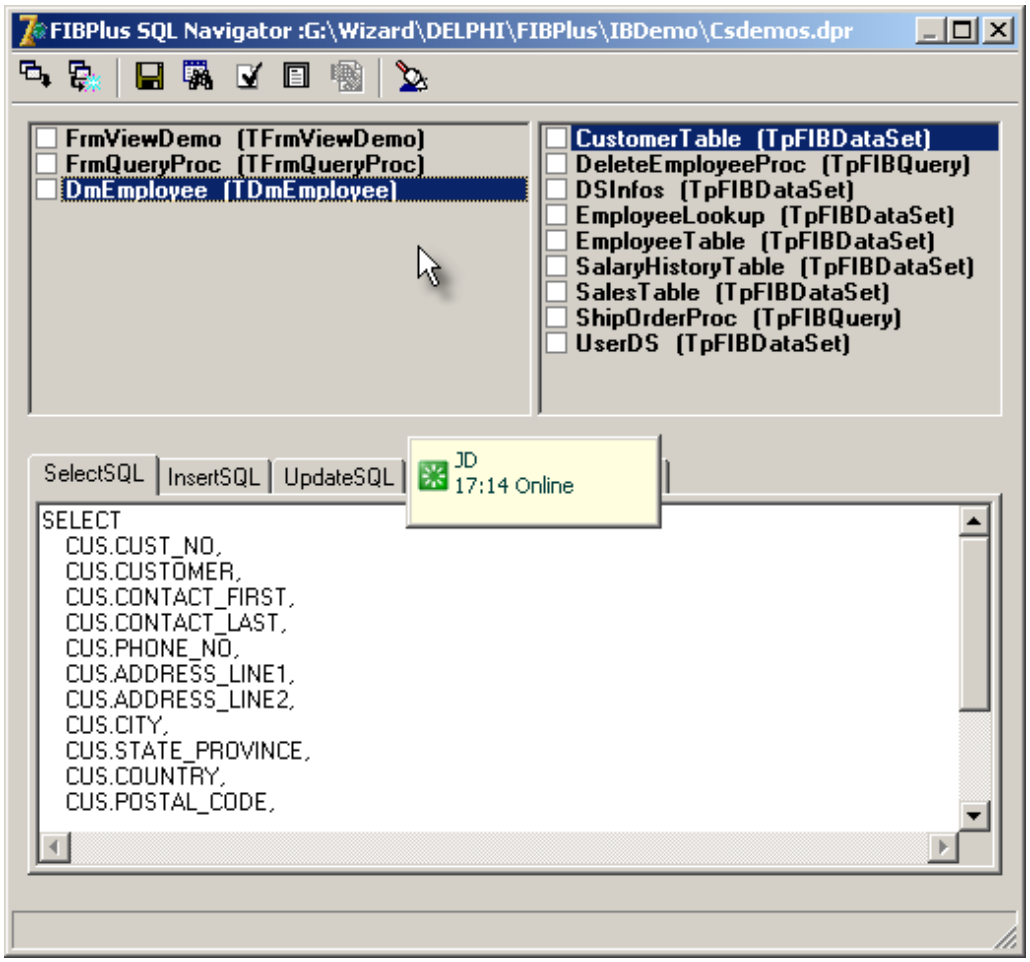


Рис. 2.54. Внешний вид SQL Navigator

SQLNavigator позволяет разработчику сосредоточиться на написании и анализе SQL-кода в приложении. Нажмите кнопку Scan all forms of active project. SQLNavigator переберет все формы приложения и выделит из них те, которые содержат компоненты FIBPlus для работы с SQL: TpFIBDataSet, TpFIBQuery, TpFIBUpdateObject и TpFIBStoredProc. Нажмите в списке на любую из обнаруженных форм. Список справа будет заполнен компонентами, обнаруженными на этой форме. Нажатие на любой из компонентов позволит нам посмотреть соответствующие свойства, в которых содержится SQL-код. Для компонентов класса

TpFIBDataSet будут выведены свойства: SelectSQL, InsertSQL, UpdateSQL, DeleteSQL и RefreshSQL. Для компонентов TpFIBQuery, TpFIBUpdateObject и TpFIBStoredProc будет выведено значение свойства SQL.

Вы можете изменить любое свойство напрямую из SQLNavigator, и новое значение будет сохранено. SQLNavigator позволяет делать операции с группами компонентов. Для этого достаточно пометить соответствующие компоненты или даже формы (рис. 2.55).

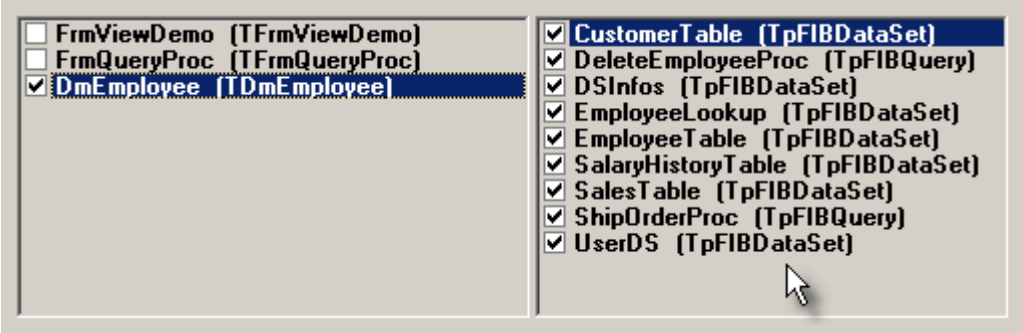


Рис. 2.55. Выделение группы компонентов при помощи SQL Navigator

Теперь мы можем сохранить значения выделенных свойств во внешнем файле при помощи кнопки Save selected SQLs или проверить их корректность прямо в SQLNavigator при помощи кнопки Check selected SQLs. Записанный файл с выделенными запросами можно использовать для дальнейшего анализа при помощи специализированных инструментов.

Вы можете использовать SQLNavigator для поиска текста в SQL в рамках всего проекта. Например, на рис. 2.56 вы можете видеть все свойства в проекте, которые содержат строку "ID".

При помощи двойного нажатия на каждом найденном элементе SQLNavigator выберет компонент и свойство, чтобы разработчик мог редактировать SQL.

Таким образом, SQLNavigator представляется очень эффективным инструментом для работы с SQL-кодом в клиентской части приложения базы данных, пожалуй, единственным в своем роде.

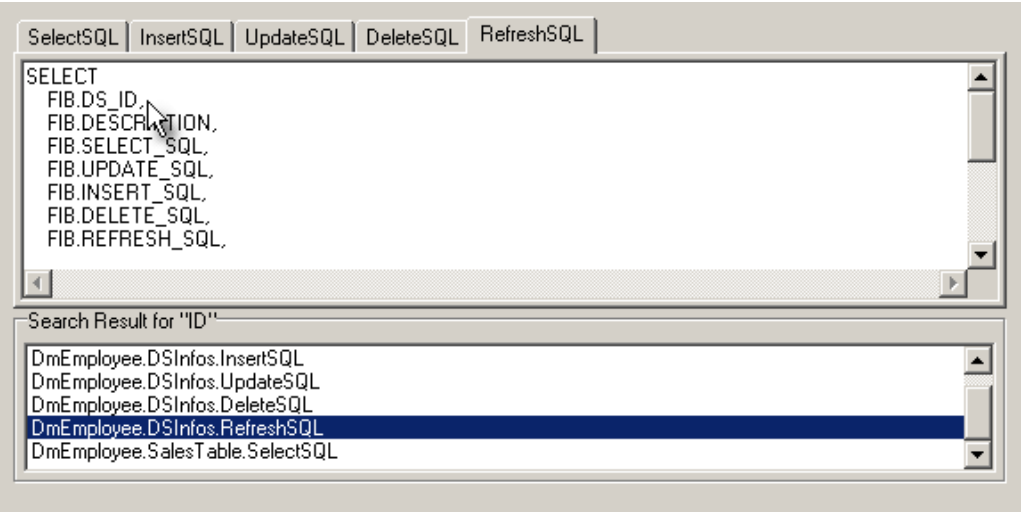


Рис. 2.56. Результат поиска текста в запросах FIBPlus

Специальные возможности FIBPlus

Обработка потери подключения к базе данных

Корректная обработка потери подключения к базе данных является одной из проблем при разработке устойчивых приложений вообще и, пожалуй, самым важным вопросом при разработке приложения, которое будет работать в условиях нестабильного канала связи (в частности, при использовании dial-up).

FIBPlus предоставляет разработчику полный набор средств для обработки потери подключения: возможность аккуратно закрыть все приложение, "закрыть" (т. е. деактивировать все соответствующие компоненты) само подключение на уровне приложения или попробовать восстановить подключение без закрытия запросов.

Ключевым компонентом для обработки ситуации является TpFIBErrorHandler. Формально, потерю подключения мы будем обрабатывать при помощи компонента TpFIBDatabase в обработчике события OnLostConnect, однако без "глубокого" перехвата в TpFIBErrorHandler мы не сможем избавиться от лишних сообщений о потере подсоединения.

Итак, попробуем создать простое приложение, позволяющее редактировать данные в TDBGrid и обрабатывать потерю подключения к базе данных тремя способами, про которые уже было упомянуто (рис. 2.57).

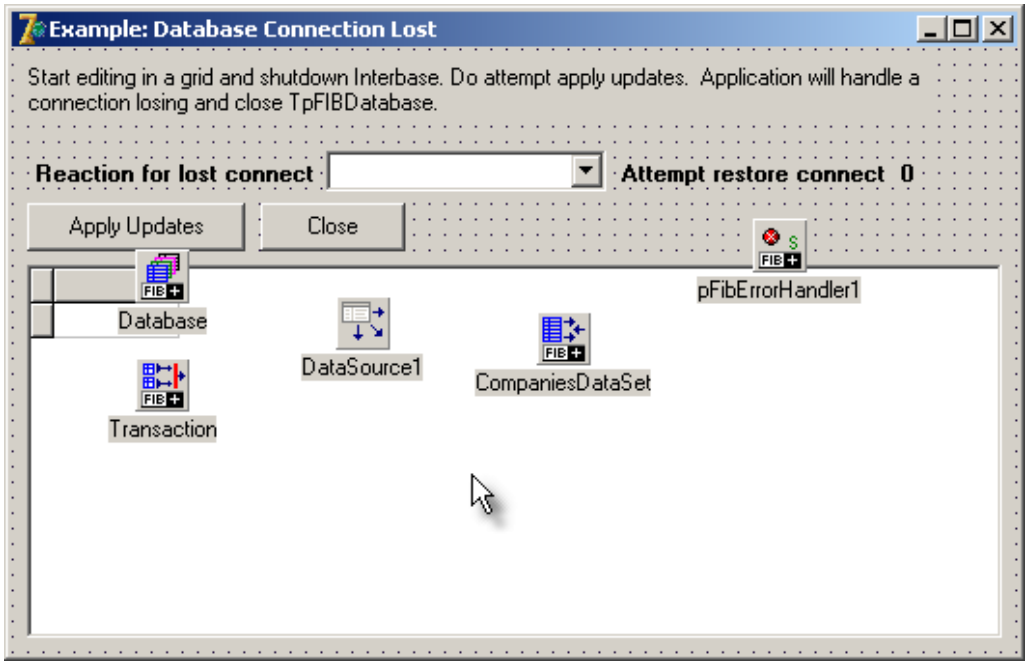


Рис. 2.57. Внешний вид главной формы примера ConnectionLost

Заполним SelectSQL для CompaniesDataSet: select * from "Companies" и предоставим CompaniesDataSet самостоятельно генерировать модифицирующие запросы (рис. 2.58).

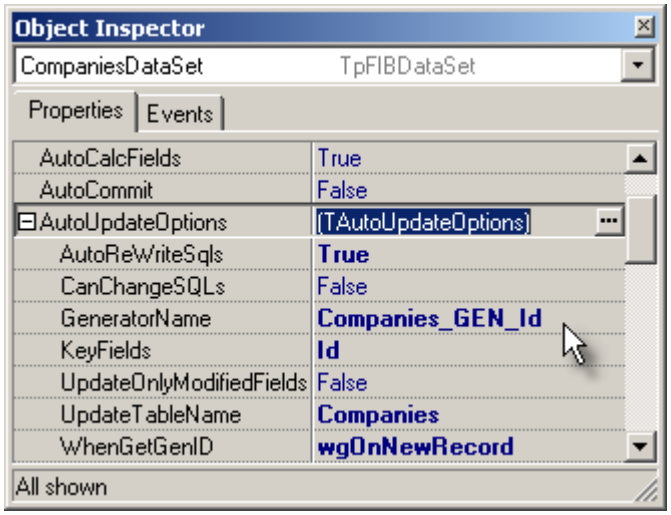


Рис. 2.58. Свойства AutoUpdateOptions компонента CompaniesDataSet

Включим также режим CachedUpdates (CompaniesDataSet.CachedUpdates := True), чтобы корректно обрабатывать восстановление подключения без потери данных.

Компонент cmbKindOnLost: TComboBox будет содержать список возможных реакций на потерю подключения, чтобы мы могли во время выполнения приложения попробовать все возможности:

```
Close pFIBDataBase
Terminate application
Restore connect
```

Теперь обратим внимание на компонент pFIBErrorHandler1 (рис. 2.59).

TpFIBErrorHandler обрабатывает "особым" образом два типа ошибок: пользовательские исключения и потерю подключения к базе данных. В компонент заложена также возможность обработки ошибок, связанных с нарушением ссылочной целостности, однако в существующих версиях FIBPlus данная функция еще не реализована.

TpFIBErrorHandler имеет только одно событие – OnFIBErrorEvent, обработка которого позволит нам обработать, в частности, потерю подключения к базе данных:

```
procedure TForm1.pFibErrorHandler1FIBErrorEvent(Sender:
TObject;
  ErrorValue: EFIBError; KindIBError: TKindIBError; var
DoRaise: Boolean);
begin
  if KindIBError = keLostConnect then begin
    DoRaise := false;
```

```
    Abort;
  end;
end;
```

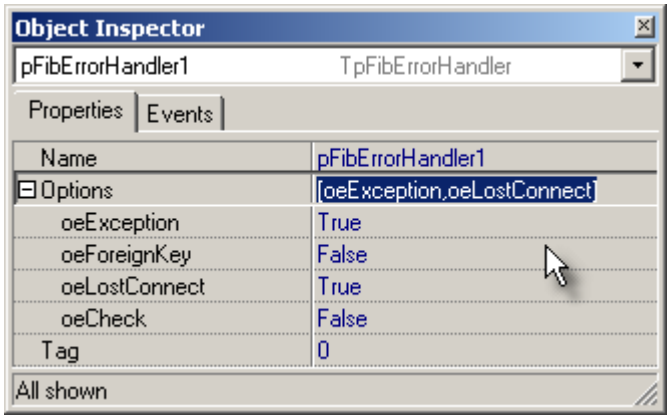


Рис. 2.59. Свойства компонента pFIBErrorHandler1

Вот в общем-то и весь обработчик – мы просто запрещаем вывод стандартного сообщения о потере подключения. На практике вы сможете использовать OnFIBErrorEvent для более сложной обработки разного рода исключительных ситуаций, используя значение параметра ErrorValue. Для нашего случая важно также знать что, кроме срабатывания OnFIBErrorEvent, TpFIBErrorHandler инициирует возникновение OnLostConnection у компонентов TpFIBDatabase. Именно здесь мы и сосредоточим основную обработку потери подсоединения:

```
procedure TForm1.DatabaseLostConnect(Database: TFIBDatabase; E:
EFIBError;
  var Actions: TOnLostConnectActions);
begin
  AttemptRest := 0;
  case cmbKindOnLost.ItemIndex of
   0: begin
       Actions := laCloseConnect;
       MessageDlg('Connection lost. TpFIBDatabase will be
closed!',
        mtInformation, [mbOk], 0
       );
      end;
   1: begin
      Actions := laTerminateApp;
      MessageDlg('Connection lost. Application will be
closed!',
       mtInformation, [mbOk], 0
      );
     end;
   2: Actions := laWaitRestore;
  end;
end;
```

Смысл обработчика очевиден – в зависимости от выбранного пользователем значения компонента cmbKindOnLost наше приложение либо "закрывает" активное подключение и все соответствующие компоненты, либо закрывает все приложение, либо включает режим восстановления подсоединения. Первые два случая очевидны и в общем-то не требуют никаких дополнительных шагов, кроме установления значения параметра Actions. Более подробно мы остановимся на восстановлении подключения.

Поскольку CompaniesDataSet находится в режиме CachedUpdates, то потеря подключения не влияет на возможность редактирования данных пользователем, поскольку все изменения будут сохраняться в локальном буфере компонента CompaniesDataSet. Таким образом, в задачу TpFIBDataBase входит только одно: периодически пытаться восстановить подключение и сообщать об удачных и неудачных попытках. Когда подсоединение будет восстановлено, мы просто применим "отложенные" изменения к данным в базе данных. Конечно, никто не гарантирует, что отложенные команды смогут быть выполнены сервером, поскольку на момент восстановления подключения наши локальные данные могут совершенно потерять актуальность.

Итак, напишем два простых обработчика. Первый для события DataBase.OnErrorRestoreConnect:

```
procedure TForm1.DatabaseErrorRestoreConnect(Database: TFIB-
Database;
  E: EFIBError; var Actions: TOnLostConnectActions);
begin
  Inc(AttemptRest);
  Label4.Caption := IntToStr(AttemptRest);
  Label4.Refresh;
end;
```

Компонент Label4 будет показывать счетчик попыток восстановления подсоединения. Второе событие, которое нас интересует, – AfterRestoreConnect:

```
procedure TForm1.DatabaseAfterRestoreConnect;
begin
  MessageDlg('Connection restored. You can apply cached
updates',
       mtInformation, [mbOk], 0
  );
end;
```

Как только подключение к базе восстановлено, мы получим сообщение и сможем применить все сделанные изменения.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with CompaniesDataSet do
    try
      if not DataBase.Connected then begin
         try
           DataBase.Connected := True;
         except
           MessageDlg('Can''t restore connect',
```

```
            mtInformation, [mbOk], 0
           );
          Exit;
        end
      end;

      if not Transaction.Active then Transac-
tion.StartTransaction;

      ApplyUpdToBase;
      Transaction.CommitRetaining;
      CommitUpdToCach;

    except
      if Transaction.Active then Transaction.RollBack;
    end;
end;
```

На практике вы можете автоматически вызывать процедуру применения данных сразу из обработчика AfterRestoreConnect. В самой процедуре следует обратить внимание на методы ApplyUpdToBase и CommitUpdToCach. В отличие от стандартного метода ApplyUpdates, наследованного от TDataSet, который "не замечает" скрытые в результате локальной фильтрации записи (более подробно этот вопрос будет рассмотрен в разделе "Локальная фильтрация"), методы ApplyUpdToBase и CommitUpdToCach позволяют обойти эту ошибку VCL.

Эмуляция Boolean-полей

Вы уже знаете, что InterBase не поддерживает логического типа данных. Чем бы это ни было вызвано, нам остается только огорчиться этим фактом, поскольку на практике логический тип очень удобен. Разработчики, использующие InterBase, вынуждены заменять его другими типами, вводя ограничение на множество допустимых значений. Как правило, в базе данных создается соответствующий домен одного из двух видов:

```
CREATE DOMAIN TBOOLEAN_CHAR AS CHAR(1)
DEFAULT 'F' NOT NULL
CHECK (VALUE IN ('F', 'T'))

CREATE DOMAIN TBOOLEAN_INT AS INTEGER
DEFAULT 0 NOT NULL
CHECK (VALUE IN (0, 1))
```

При использовании любой библиотеки компонент для доступа к InterBase оба способа совершенно равноценны, поскольку для полей, созданных с такими доменами, в приложении все равно не создаются Boolean-поля. То есть если мы добавим некоторое логическое поле к нашей таблице:

```
ALTER TABLE "Categories"
ADD IS_ACTIVE TBOOLEAN_INT
NOT NULL
```

А потом, используя, например, компоненты IBX, сделаем выборку при помощи компонента TIBDataSet:

```
SELECT "Categories"."Name", "Categories".IS_ACTIVE
FROM "Categories"
```

то в результате компонент создаст два внутренних компонента для полей:

- TIBStringField для поля "Name" и
- TintergerField для поля "IS_ACTIVE"

Последнее совершенно верно, поскольку формально поле "IS_ACTIVE" не является логическим. Таким образом, все визуальные компоненты типа TDBGrid и его расширенные аналоги от сторонних производителей не будут обрабатывать данное поле так, как нам бы хотелось. Например, если даже компонент умеет рисовать "галочки" для Boolean-полей, значения которых равно True, то поскольку он будет "видеть" всего лишь целочисленное поле, то и выводить он будет "0" и "1". Разумеется, если мы напишем соответствующие обработчики событий, то сможем добиться более или менее сносного отображения логических величин для поля TIntergerField, однако FIBPlus предоставляет гораздо более простое и качественное решение.

Фактически оно состоит только в том, что нам необходимо добавить ключ poUseBooleanField в свойстве PrepareOptions (рис. 2.60).

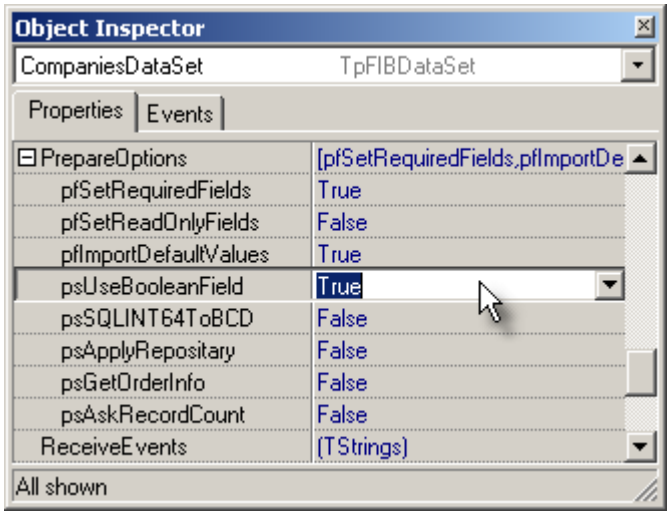


Рис. 2.60. Использование PrepareOptions для эмуляции Boolean-полей

После этого поля, созданные на основе целочисленного домена, в названии которого присутствует слово "boolean", будут считать логическими, и для них будут создаваться экземпляры TFIBBooleanField. Данный класс является прямым потомком класса TBooleanField и является полноценным логическим полем. Любые визуальные компоненты для отображения данных будут работать с такими полями как с логическими, т. е. используя свойство AsBoolean. Вам же не придется писать для этого никакой дополнительный код.

Поддержка array-полей. Пример использования TpFIBUpdateObject и TDataSetContainer

InterBase с самых ранних версий позволял описывать в таблицах многомерные поля-массивы, делая хранение специализированных данных максимально удобным. Вы наверняка согласитесь, что матрицу проще всего хранить и обрабатывать в виде матрицы, а не раскладывать ее по отдельным полям и даже таблицам из-за ограничений реляционной модели. Тем не менее поскольку array-поля не поддерживаются стандартом SQL, то и работа с такими полями на уровне SQL-запросов крайне затруднена. Фактически вы можете использовать массивы только поэлементно и только в операциях чтения. Чтобы изменить значения array-поля, необходимо использовать специальные команды Interbase API. FIBPlus позволяет обойтись без подобных сложностей, взяв на себя всю рутину, связанную с array-полями.

Мы продемонстрируем, как работать с array-полями при помощи FIBPlus на примере DemoArray5, входящем в стандартную поставку FIBPlus. Пример демонстрирует два варианта использования array-полей. Первый способ позволяет редактировать array-поле при помощи специальных методов ArrayFieldValue и SetArrayValue и работать с таким полем как с единой структурой (рис. 2.61).

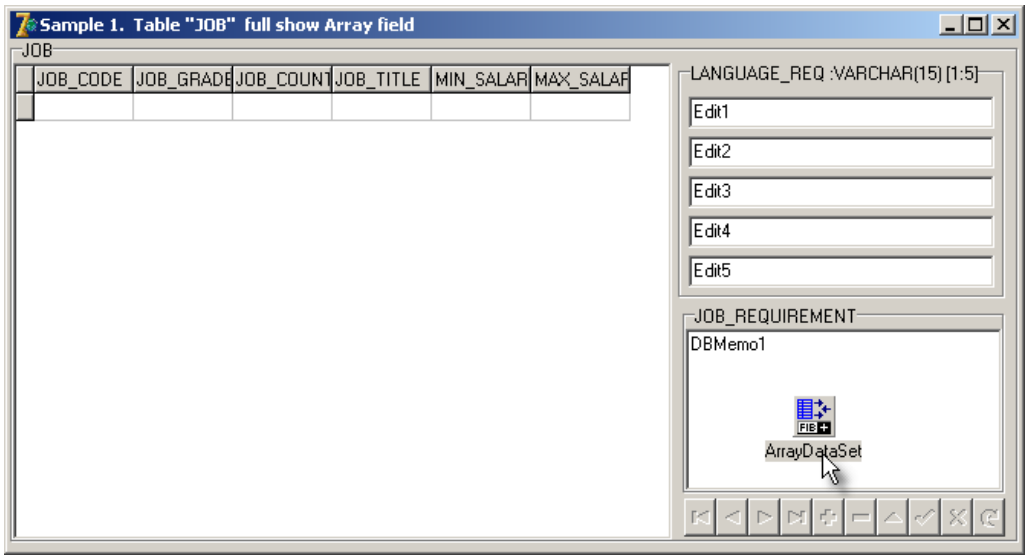


Рис. 2.61. Внешний вид формы примера DemoArray5. Первый вариант использования array-полей

Рассмотрим запросы, заданные в соответствующих свойствах ArrayDataSet:
SelectSQL:

```
SELECT
    JOB.JOB_CODE,
    JOB.JOB_GRADE,
    JOB.JOB_COUNTRY,
    JOB.JOB_TITLE,
```

```
    JOB.MIN_SALARY,
    JOB.MAX_SALARY,
    JOB.JOB_REQUIREMENT,
    JOB.LANGUAGE_REQ
FROM
    JOB JOB
ORDER BY 1,2,3
```

UpdateSQL:

```
UPDATE JOB SET
    JOB_CODE = ?JOB_CODE,
    JOB_GRADE = ?JOB_GRADE,
    JOB_COUNTRY = ?JOB_COUNTRY,
    JOB_TITLE = ?JOB_TITLE,
    MIN_SALARY = ?MIN_SALARY,
    MAX_SALARY = ?MAX_SALARY,
    JOB_REQUIREMENT = ?JOB_REQUIREMENT,
    JOB.LANGUAGE_REQ= ?LANGUAGE_REQ
 WHERE
     JOB_CODE = ?OLD_JOB_CODE
     and JOB_GRADE = ?OLD_JOB_GRADE
     and JOB_COUNTRY = ?OLD_JOB_COUNTRY
```

InsertSQL:

```
INSERT INTO JOB(
    JOB_CODE,
    JOB_GRADE,
    JOB_COUNTRY,
    JOB_TITLE,
    MIN_SALARY,
    MAX_SALARY,
    JOB_REQUIREMENT,
    JOB.LANGUAGE_REQ
)
VALUES(
    ?JOB_CODE,
    ?JOB_GRADE,
    ?JOB_COUNTRY,
    ?JOB_TITLE,
    ?MIN_SALARY,
    ?MAX_SALARY,
    ?JOB_REQUIREMENT,
    ?LANGUAGE_REQ
)
```

DeleteSQL:

```
DELETE FROM JOB
 WHERE
     JOB_CODE = ?OLD_JOB_CODE
     and JOB_GRADE = ?OLD_JOB_GRADE
     and JOB_COUNTRY = ?OLD_JOB_COUNTRY
```

RefreshSQL:

```
SELECT
    JOB.JOB_CODE,
    JOB.JOB_GRADE,
    JOB.JOB_COUNTRY,
    JOB.JOB_TITLE,
    JOB.MIN_SALARY,
    JOB.MAX_SALARY,
    JOB.JOB_REQUIREMENT,
    JOB.LANGUAGE_REQ
FROM
    JOB JOB
 WHERE
    (
     JOB.JOB_CODE = ?OLD_JOB_CODE
     and JOB.JOB_GRADE = ?OLD_JOB_GRADE
     and JOB.JOB_COUNTRY = ?OLD_JOB_COUNTRY
    )
```

Поле LANGUAGE_REQ является массивом (LANGUAGE_REQ VARCHAR(15) [1:5]) и, как видно из запросов, обрабатывается целиком, а не поэлементно. С одной стороны это удобно, но не позволяет использовать для редактирования таких полей специализированные визуальные компоненты типа TDBGrid. Если мы используем array-поля для хранения значительных массивов данных, мы в любом случае будем использовать "ручную" обработку данных, не прибегая к помощи визуальных компонент. Однако для наглядности примера мы позволим редактировать элементы массива в компонентах TEdit.

Фактически нам понадобится написать только два основных обработчика для событий: BeforePost и OnPostError:

```
procedure TForm1.ArrayDataSetBeforePost(DataSet: TDataSet);
begin
  with ArrayDataSet do begin
    SetArrayValue(FieldByName('LANGUAGE_REQ'),
      VarArrayOf([
         Edit1.Text,
         Edit2.Text,
         Edit3.Text,
         Edit4.Text,
         Edit5.Text
       ]));
  end;
end;

procedure TForm1.ArrayDataSetPostError(DataSet: TDataSet;
  E: EDatabaseError; var Action: TDataAction);
begin
  Action := daAbort;
  MessageDlg('Error!', mtError, [mbOk], 0);
  ArrayDataSet.Refresh;
end;
```

Метод SetArrayValue позволяет задать все элементы поля в виде массива. Важным моментом является обработчик ошибки ArrayDataSetPostError. В случае неудачной операции Update или Insert необходимо восстанавливать внутренний идентификатор массива у редактируемой записи. Это правило диктуется функциями InterBase API, и мы должны их придерживаться. Для восстановления идентификатора необходимо получить значения полей текущей записи заново, что и делается при помощи явного вызова метода Refresh.

Для автоматического заполнения визуальных компонентов значениями элементов массива мы можем написать обработчик события AfterScroll:

```
procedure TForm1.ArrayDataSetAfterScroll(DataSet: TDataSet);
var v: Variant;
begin
  with ArrayDataSet do try
    FInShowArrays := true;
    v := ArrayFieldValue(FieldByName('LANGUAGE_REQ'));
    Edit1.Text := VarToStr(v[1]);
    Edit2.Text := VarToStr(v[2]);
    Edit3.Text := VarToStr(v[3]);
    Edit4.Text := VarToStr(v[4]);
    Edit5.Text := VarToStr(v[5]);
  finally
    FInShowArrays:=false;
  end;
end;
```

Флаг FInShowArrays используется в примере для того, чтобы не включать режим редактирования записи при обычной навигации.

> Рекомендуется иметь под руками полный текст примера, который доступен на сайте http://www.fibplus.net. В книге мы указываем только те части примера, которые имеют непосредственное и наибольшее значение для работы с array-полями. Однако без полного текста некоторые части исходного текста могут показаться не до конца наглядными.
>
> Данный способ работы с array-полями нельзя применять в режиме CachedUpdates.

Второй способ работы с array-полями позволяет использовать их в "живых" запросах и редактировать при помощи стандартных визуальных компонентов (рис. 2.62).

Рассмотрим запросы, указанные в свойствах ArrayDataSet:

SelectSQL:

```
SELECT
    JOB.JOB_CODE,
    JOB.JOB_GRADE,
    JOB.JOB_COUNTRY,
    JOB.JOB_TITLE,
    JOB.LANGUAGE_REQ[1] LQ1,
    JOB.LANGUAGE_REQ[2] LQ2
FROM
    JOB JOB
ORDER BY 1,2,3
```

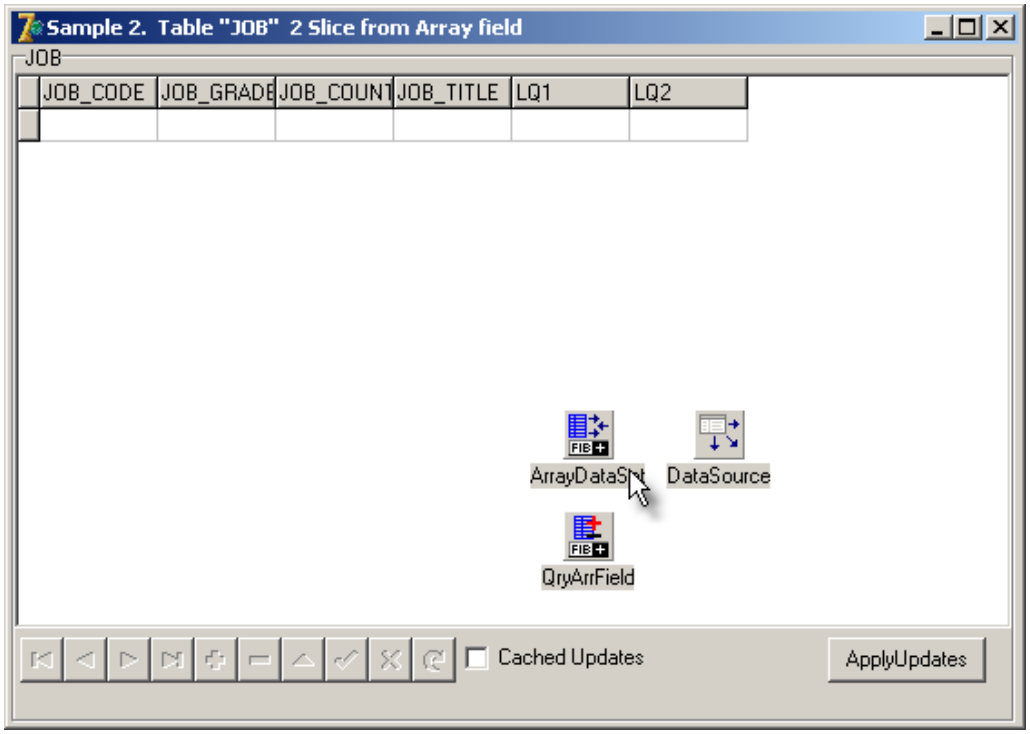


Рис. 2.62. Внешний вид формы примера DemoArray5. Второй вариант использования array-полей

UpdateSQL:

```
UPDATE JOB SET
    JOB_TITLE = ?JOB_TITLE,
    JOB.JOB_GRADE=?JOB_GRADE,
    JOB.JOB_COUNTRY=?JOB_COUNTRY,
    LANGUAGE_REQ = ?LQ
 WHERE
             JOB_CODE = ?OLD_JOB_CODE
     and JOB_GRADE = ?OLD_JOB_GRADE
     and JOB_COUNTRY = ?OLD_JOB_COUNTRY
```

InsertSQL:

```
INSERT INTO JOB(
    JOB_CODE,
    JOB_GRADE,
    JOB_COUNTRY,
    JOB_TITLE,
    LANGUAGE_REQ
)
VALUES(
    ?JOB_CODE,
    ?JOB_GRADE,
    ?JOB_COUNTRY,
    ?JOB_TITLE,
```

```
    ?LQ
)
```

DeleteSQL:

```
DELETE FROM JOB
 WHERE
            JOB_CODE = ?OLD_JOB_CODE
     and JOB_GRADE = ?OLD_JOB_GRADE
     and JOB_COUNTRY = ?OLD_JOB_COUNTRY
```

RefreshSQL:

```
SELECT
    JOB.JOB_CODE,
    JOB.JOB_GRADE,
    JOB.JOB_COUNTRY,
    JOB.JOB_TITLE,
    JOB.LANGUAGE_REQ[1] LQ1,
    JOB.LANGUAGE_REQ[2] LQ2
FROM
    JOB JOB
 WHERE
    (
            JOB.JOB_CODE = ?OLD_JOB_CODE
     and JOB.JOB_GRADE = ?OLD_JOB_GRADE
     and JOB.JOB_COUNTRY = ?OLD_JOB_COUNTRY
    )
```

На этот раз мы выбираем только два элемента из нашего поля-массива. Обратите внимание: несмотря на то что в выбирающем запросе мы явным образом выделяем два элемента массива, а в модифицирующих запросах мы обновляем поле целиком. На самом деле, это, конечно, не совсем так, однако данный синтаксис наиболее удобен и близок к естественному SQL-запросу, несмотря на то что во внутренней реализации FIBPlus использует специальные функции работы с массивами. Обратим внимание на компонент QryArrField: TpFIBUpdateObject. Именно он позволит нам правильно сформировать значение параметра "LQ" в модифицирующих запросах. Разумеется, для этой цели вполне бы подошел и простой TpFIBQuery, однако удобство TpFIBUpdateObject заключается в первую очередь в том, что он выполняет запрос автоматически и сам подставляет туда нужные значения параметров в зависимости от значений TpFIBDataSet. Вот запрос, который выполняет QryArrField:

```
SELECT
    JOB.LANGUAGE_REQ,
    JOB.JOB_CODE
FROM
    JOB JOB
WHERE
    JOB.JOB_CODE=?OLD_JOB_CODE       and
    JOB.JOB_GRADE=?OLD_JOB_GRADE     and
    JOB.JOB_COUNTRY=?OLD_JOB_COUNTRY
```

Обратим внимание на свойства QryArrField (рис. 2.63)

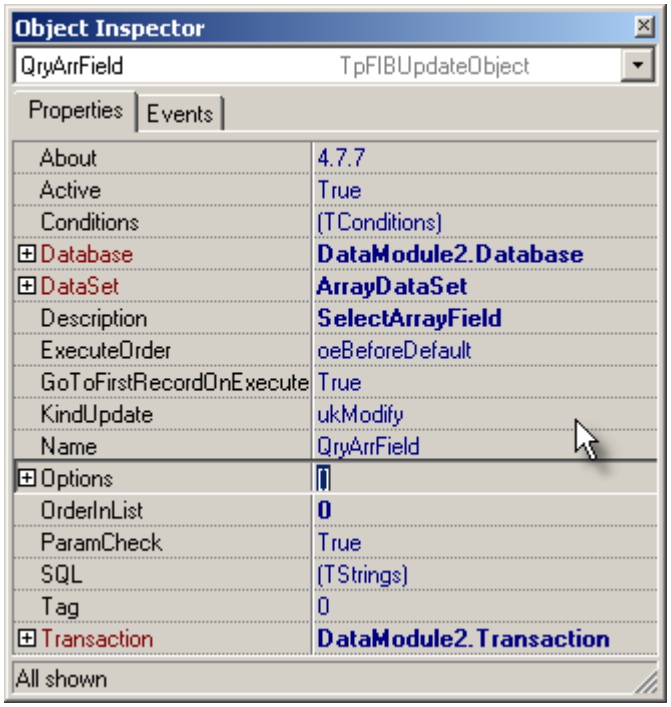


Рис. 2.63. Свойство компонента QryArrField

Обработчик QryArrField.AfterExecute:

```
procedure TForm2.QryArrFieldAfterExecute(Sender: TObject);
var v: Variant;
begin
  v := QryArrField.Fields[0].GetArrayValues;
  with ArrayDataSet do begin
    v[1] := FieldByName('LQ1').AsString;
    v[2] := FieldByName('LQ2').AsString;
    QryArrField.Fields[0].SetArrayValue(v);
    QUpdate.Params.ByName['LQ'].AsQuad :=
QryArrField.Fields[0].AsQuad;
  end;
  QryArrField.Close;
end;
```

Теперь все становится совершенно очевидным. Поскольку свойство QryArrField.KindUpdate равно ukModify, то QryArrField выполняет запрос при изменении записи в ArrayDataSet, а поскольку свойство QryArrField.ExecuteOrder равно eoBeforeDefault, то запрос (QryArrField.SQL) выполняется до того, как ArrayDataSet выполнит свой собственный UpdateSQL. В обработчике QryArrField.AfterExecute мы всего лишь получаем заново все текущие элементы массива из базы данных, подменяем два из них новыми значениями, которые указал пользователь, и задаем значение параметра для ArrayDataSet.UpdateSQL. Это означает, что при выполнении ArrayDataSet.UpdateSQL формально будут

обновлены все пять элементов массива, но фактически изменены значения только двух элементов, которые изменил пользователь в TDBGrid.

Как видите, работа с массивами достаточно проста, поскольку все основные сложности решают компоненты FIBPlus. Хотелось бы также рассмотреть еще один специализированный компонент, входящий в FIBPlus. Пример DemoArray демонстрирует работу с TDataSetContainer, использованным для синхронизации значений двух TpFIBDataSet, редактирующих наше array-поле (рис. 2.64).

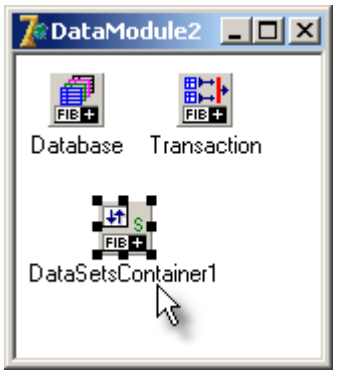


Рис. 2.64. Использование TDataSetContainer

Компонент DataSetContainer1 помещен вместе с Database и Transaction на DataModule в нашем приложении. Оба компонента ArrayDataSet из разных форм нашего приложения ссылаются на DataSetContainer1 при помощи свойства Container (рис. 2.65).

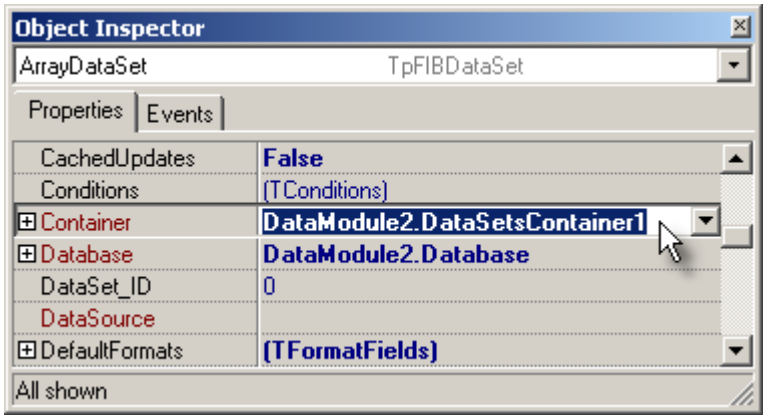


Рис. 2.65. Подключение компонентов TpFIBDataSet к DataSetContainer1

Компонент TDataSetContainer позволяет централизованно обрабатывать события от разных компонентов TpFIBDataSet, а также (расширяя, таким образом, список стандартных событий) посылать им сообщения, при получении которых они могут производить какие-то дополнительные действия. В нашем примере

DataSetContainer1 имеет обработчики двух событий – OnDataSetEvent и OnUserEvent:

```
procedure TDataModule2.DataSetsContainer1DataSetEvent(DataSet:
TDataSet;
  Event: TKindDataSetEvent);
var Info: string;
begin
  if Event = deAfterPost then
    if DataSet.Owner.Name = 'Form1' then
        DataSetsContainer1.NotifyDataSets(DataSet,
          'Form2.ArrayDataSet', 'JOB_TABLE_CHANGED', Info)
    else
      if DataSet.Owner.Name = 'Form2' then
        DataSetsContainer1.NotifyDataSets(DataSet,
          'Form1.ArrayDataSet', 'JOB_TABLE_CHANGED', Info);
end;
procedure TDataModule2.DataSetsContainer1UserEvent(Sender:
TObject;
  Receiver: TDataSet; const EventName: String; var Info:
String);
begin
  if EventName = 'JOB_TABLE_CHANGED' then begin
    with TpFIBDataSet(Sender) do
      if (not CachedUpdates) and
         (not TpFIBDataSet(Receiver).CachedUpdates) then
       if TpFIB-
DataSet(Receiver).Locate('JOB_CODE;JOB_GRADE;JOB_COUNTRY',
          varArrayOf([
            FieldByName('JOB_CODE').AsString,
            FieldByName('JOB_GRADE').AsString,
            FieldByName('JOB_COUNTRY').AsString
          ]), []) then TpFIBDataSet(Receiver).Refresh
  end;
end;
```

Смысл действий сводится к следующему: после изменения записи в одном из наших компонентов ArrayDataSet происходит событие AfterPost. Поскольку DataSetContainer1 перехватывает все события у подчиненных компонентов, то срабатывает обработчик OnDataSetEvent. Параметр Event равен deAfterPost, а параметр DataSet ссылается на тот компонент, в котором произошло изменение записи. При помощи вызова метода NotifyDataSets DataSetContainer1 посылает сообщение оставшемуся компоненту ArrayDataSet о том, что произошло изменение записи. Поскольку оба компонента на самом деле редактируют одну и ту же таблицу, то желательно синхронизировать изменения. Синхронизация происходит в обработчике события OnUserEvent, то есть при получении "извещения" об изменении какого-либо из ArrayDataSet.

Если получено сообщение "JOB_TABLE_CHANGED" и ни в одном из наших двух ArrayDataSet не включен режим CachedUpdates (в этом случае Refresh просто ничего не даст), то мы позиционируемся на соответствующую запись и вы-

зываем для нее метод Refresh, обновив, таким образом, запись в одном ArrayDataSet после того, как она была изменена в другом ArrayDataSet.

Данная технология является совершенно уникальной: использование DataSetContainer позволяет делать чрезвычайно гибкие и в то же время прозрачные схемы обработки и синхронизации данных в нескольких компонентах TpFIBDataSet.

Работа с BLOB-полями

Достаточно часто желательно хранить в базе данных разнообразные неструктурированные данные: изображения, OLE-объекты, звук и т. д. Специально для этих целей существует специальный тип данных – BLOB. Продемонстрируем использование BLOB-полей на примере простого приложения (см. рис. 2.66), использующего следующую таблицу:

```
CREATE TABLE BIOLIFE (
    ID INTEGER NOT NULL,
    CATEGORY VARCHAR (15) character set WIN1251 collate
WIN1251,
    COMMON_NAME VARCHAR (30) character set WIN1251 collate
WIN1251,
    SPECIES_NAME VARCHAR (40) character set WIN1251 collate
WIN1251,
    LENGTH__CM_ DOUBLE PRECISION,
    LENGTH_IN DOUBLE PRECISION,
    NOTES BLOB sub_type 1 segment size 80,
    GRAPHIC BLOB sub_type 0 segment size 80);
```

Для вывода изображений, сохраненных в поле GRAPHIC, мы будем использовать стандартный компонент DBIMage1: TDBImage. Очевидно также, что запросы при работе с BLOB-полями ничем не отличаются от запросов со стандартными типами полей:

SelectSQL:

```
SELECT * FROM BIOLIFE
```

UpdateSQL:

```
UPDATE BIOLIFE Set
     ID=?NEW_ID,
     CATEGORY=?NEW_CATEGORY,
     COMMON_NAME=?NEW_COMMON_NAME,
     SPECIES_NAME=?NEW_SPECIES_NAME,
     LENGTH__CM_=?NEW_LENGTH__CM_,
     LENGTH_IN=?NEW_LENGTH_IN,
     NOTES=?NEW_NOTES,
     GRAPHIC=?NEW_GRAPHIC
WHERE ID=?OLD_ID
```

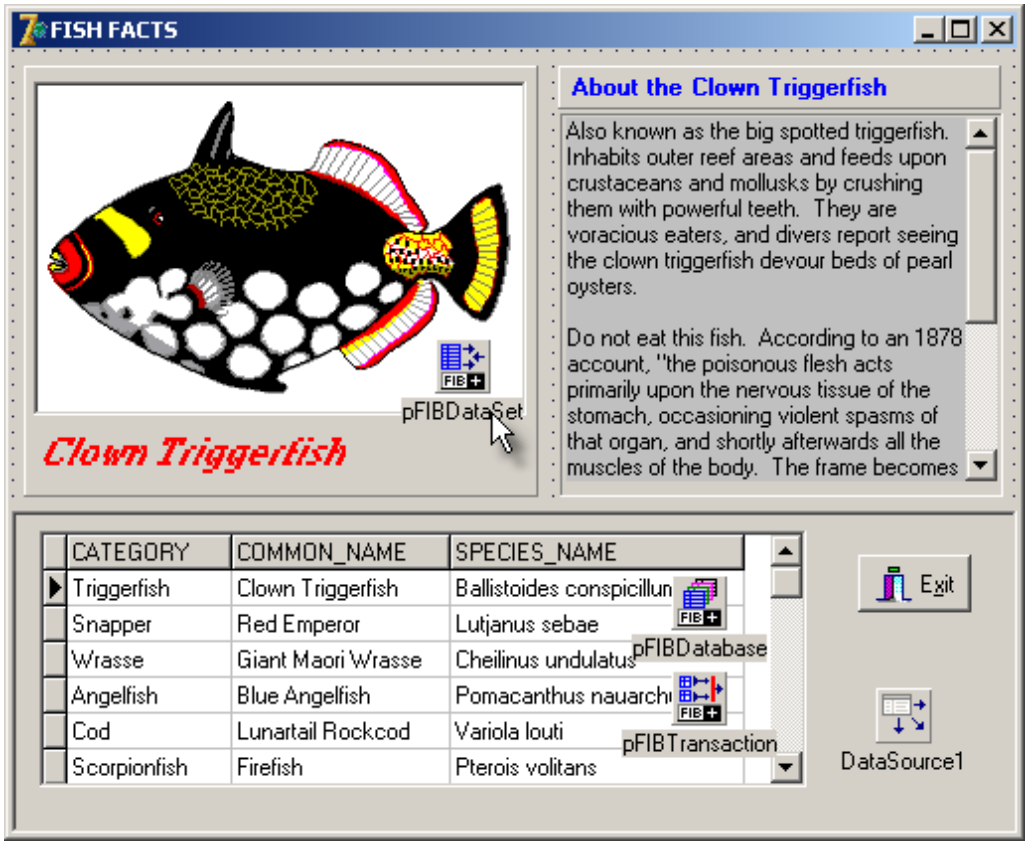


Рис. 2.66. Внешний вид формы приложения для работы с BLOB-полями

InsertSQL:

```
INSERT INTO BIOLIFE(
      ID,
      CATEGORY,
      COMMON_NAME,
      SPECIES_NAME,
      LENGTH__CM_,
      LENGTH_IN,
      NOTES,
      GRAPHIC
)
VALUES (
      ?NEW_ID,
      ?NEW_CATEGORY,
      ?NEW_COMMON_NAME,
      ?NEW_SPECIES_NAME,
      ?NEW_LENGTH__CM_,
      ?NEW_LENGTH_IN,
      ?NEW_NOTES,
      ?NEW_GRAPHIC
)
```

DeleteSQL:

```
DELETE FROM BIOLIFE
WHERE ID=?OLD_ID
```

RefreshSQL:

```
SELECT * FROM BIOLIFE
 WHERE
    ID=?OLD_ID
```

Единственным отличием от обычных типов данных является то, что для присвоения значения BLOB-параметру необходимо использовать потоки (специализированные потомки стандартного класса TStream). Например, если мы хотим сохранить в нашем поле изображение из внешнего файла, то мы можем написать следующий обработчик нажатия на кнопку:

```
procedure TMainForm.OpenBClick(Sender: TObject);
var S: TStream;
    FileS: TFileStream;
begin
  if not OpenD.Execute then exit;
  pFIBDataSet1.Edit;
  S := pFIB-
DataSet1.CreateBlobStream(pFIBDataSet1.FieldByName('GRAPHIC'),
bmReadWrite);
  try
    FileS := TFileStream.Create(OpenD.FileName, fmOpenRead);
    S.CopyFrom(FileS, FileS.Size);
  finally
    FileS.Free;
    S.Free;
    pFIBDataSet1.Post;
  end;
end;
```

Обратите внимание на важный момент – перед тем как присваивать значение BLOB-параметру, необходимо перевести pFIBDataSet в состояние редактирования данных. В данном случае это делается безусловным вызовом метода Edit. Вызов метода CreateBlobStream создает экземпляр специального внутреннего класса TFIBDSBlobStream. Скорее всего, вам не придется использовать этот класс напрямую. В нашем примере он нужен только для обмена данными между BLOB-параметром и потоком, который читает данные из файла с изображением. Параметр bmReadWrite означает, что мы собираемся изменять содержимое параметра. Поток S напрямую связан с BLOB-параметром, поэтому, копируя данные из файла (FileS) в поток S, мы фактически присваиваем значение параметру. Остается только сохранить изменения вызовом метода Post. Аналогичным образом мы можем сохранить значение BLOB-поля в некоторый внешний файл:

```
procedure TMainForm.SaveBClick(Sender: TObject);
var S: TStream;
    FileS: TFileStream;
begin
  if not SaveD.Execute then exit;
  if not pFIBDataset1.FieldByName('GRAPHIC').IsNull then begin
```

```
    S := pFIB-
Dataset1.CreateBlobStream(pFIBDataset1.FieldByName('GRAPHIC'),
bmRead);
    try
      if FileExists(SaveD.FileName) then
        FileS := TFileStream.Create(SaveD.FileName, fmOpen-
Write)
      else
        FileS := TFileStream.Create(SaveD.FileName, fmCreate);
      FileS.CopyFrom(S, S.Size);
    finally
      S.Free;
      FileS.Free;
    end;
  end;
end;
```

Обратите внимание, что в этой процедуре мы используем параметр bmRead при создании потока S. Очевидно, что для сохранения содержимого BLOB-поля в файл нам не нужно изменять само поле, поэтому мы создаем поток только для чтения. Еще более простым способом мы можем очистить содержимое BLOB-поля:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  pFIBDataSet1.Edit;
  pFIBDataSet1.FieldByName('GRAPHIC').Clear;
  pFIBDataSet1.Post;
end;
```

В этом случае даже не требуется создавать какие-либо потоки. Иногда также нужно знать, является ли BLOB-поле пустым или нет. При использовании визуальных компонентов типа TDBImage мы не можем быть в этом уверены. Согласитесь, что никто не мешает нам "нарисовать" пустую картинку и сохранить ее в BLOB-поле. В этом случае мы не сможем отличить "на глаз": есть ли какое-то изображение в BLOB-поле или нет. Однако мы можем написать обработчик события OnDataChange компонента DataSource1: TDataSource:

```
procedure TMainForm.DataSource1DataChange(Sender: TObject;
Field: TField);
begin
  CheckBox1.Checked := pFIB-
DataSet1.FieldByName('GRAPHIC').IsNull;
end;
```

Это событие вызывается, в частности, при навигации по DBGrid1; таким образом, мы всегда можем узнать, является ли текущее поле пустым или нет.

Если вы используете TpFIBQuery для работы с BLOB-полями, то общий принцип остается тем же – необходимо использовать потоки, однако, в отличие от TpFIBDataSet, вам не потребуется создавать какие-то специальные потоки. Например, мы можем написать следующую процедуру, которая сохранит в файлы все изображения из нашей таблицы:

```
pFIBQuery.SQL: SELECT * FROM BIOLIFE

procedure TMainForm.Button2Click(Sender: TObject);
```

```
var SaveFile: TFileStream;
    Index: Integer;
begin
 with pFIBQuery1 do begin
   ExecQuery;
   Index := 1;
   while not Eof do begin
     SaveFile := TFileStream.Create(IntToStr(Index) + '.bmp',
fmCreate);
     FN('GRAPHIC').SaveToStream(SaveFile);
     Next;
     inc(Index);
     SaveFile.Free;
   end;
   Close;
 end;
end;
```

Метод FN является аналогом FieldByName.

Смысл кода, приведенного выше, совершенно очевиден: мы получаем все записи из таблицы BIOLIFE, в цикле получаем от сервера очередную запись из запроса, создаем файл при помощи потока SaveFile, сохраняем в него значение поля GRAPHIC и запрашиваем следующую запись при помощи метода Next. Аналогичным образом мы могли бы присваивать значение BLOB-параметру:

```
pFIBQuery.SQL: INSERT INTO BIOLIFE (GRAPHIC) VALUES (?GRAPHIC)

procedure TMainForm.Button2Click(Sender: TObject);
var SaveFile: TFileStream;
    Index: Integer;
begin
 with pFIBQuery1 do begin
   Prepare;
   for Index := 1 to 3 do begin
     SaveFile := TFileStream.Create(IntToStr(Index) + '.bmp',
fmOpenRead);
     Params[0].LoadFromStream(SaveFile);
     SaveFile.Free;
     ExecQuery;
   end;
   Transaction.Commit;
 end;
end;
```

Данный пример вставляет три новые записи в таблицу BIOLIFE и сохраняет в них изображения из некоторых файлов "1.bmp", "2.bmp" и "3.bmp".

Поскольку в данном примере для сохранения изменений использовался метод Commit, то необходимо перезапустить приложение, чтобы увидеть вставленные записи в DBGrid1.

Локальная сортировка и локальная фильтрация

TpFIBDataSet позволяет разработчику оптимизировать работу с базой данных, используя в случае необходимости локальную сортировку данных и локальную фильтрацию. Термин "локальный" в данном случае означает, что в этих механизмах не используются дополнительные запросы к серверу. Таким образом, в определенных случаях вы можете значительно снизить сетевой трафик вашего приложения, а также увеличить скорость поиска или сортировки данных.

Локальная сортировка

Рассмотрим локальную сортировку на примере Sorting, который входит в поставку FIBPlus.

> Как и все остальные примеры, приложение Sorting вы можете найти на сайте http://www.fibplus.net.

Пример использует таблицу EMPLOYEE из базы данных Employee.gdb (рис. 2.67)

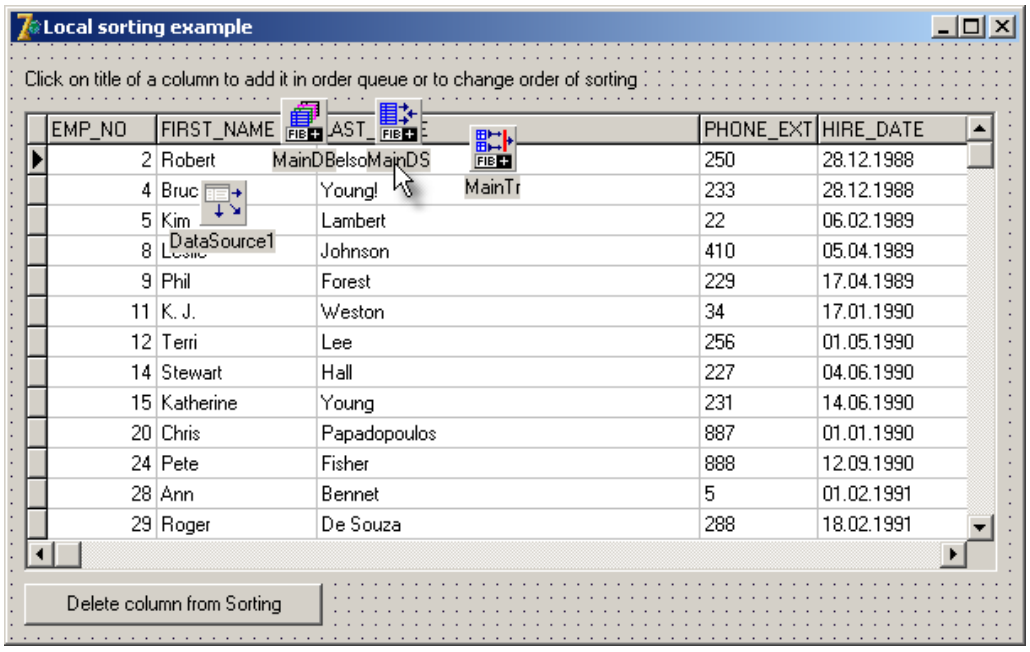


Рис. 2.67. Внешний вид формы приложения, демонстрирующего локальную сортировку в TpFIBDataSet

```
MainDS.SelectSQL: SELECT * FROM EMPLOYEE
```

Все записи, полученные от сервера в результате запроса, сохраняются в локальном буфере компонента TpFIBDataSet.

Это одна из причин, из-за которой не рекомендуется использовать TpFIBDataSet для слишком больших выборок, – вам просто может не хватить оперативной памяти и работа приложения замедлится.

Предположим, что пользователь хочет отсортировать записи по полю FIRST_NAME. Очевидно, что мы можем сформировать новый запрос:

```
SELECT * FROM EMPLOYEE
ORDER BY FIRST_NAME
```

Чтобы получить новый порядок записей, необходимо переоткрыть наш запрос. Это работает, но означает, что мы будем повторно получать от сервера все записи, несмотря на то что нам нужно было всего лишь поменять их визуальный порядок. Если предполагать, что нашим приложением могут пользоваться одновременно несколько пользователей, то затраты на подобные "сортировки" могут оказаться довольно значительными. Тем не менее выход существует, поскольку TpFIBDataSet позволяет сортировать данные внутри своего локального буфера, т. е. без необходимости получения всех записей заново.

Для этого мы должны воспользоваться одним из возможных вариантов метода DoSort или DoSortEx:

```
    procedure DoSort(Fields: array of const; Ordering: array of
Boolean); virtual;
    procedure DoSortEx(Fields: array of integer; Ordering: ar-
ray of Boolean); overload;
    procedure DoSortEx(Fields: TStrings; Ordering: array of
Boolean); overload;
```

Метод DoSortEx доступен в FIBPlus начиная с версии Delphi 4.

Первый параметр всех трех процедур – это список полей, по которым мы хотим отсортировать данные. В случае DoSort это могут быть названия полей или номера полей. Первый вариант DoSortEx позволяет нам использовать только список с номерами полей, а второй вариант DoSortEx предполагает, что мы заполнили список Fields названиями полей. Параметр Ordering во всех трех случаях указывает направление сортировки по каждому из полей. Таким образом, мы можем отсортировать наш запрос по полю FIRST_NAME следующим образом:

```
DoSort(['FIRSN_NAME'], [True]);
```

Или:

```
DoSortEx([1], [True]);
```

[True] означает, что поле сортируется по возрастанию (ASCENDING).

В сущности, использование этих методов очевидно, однако иногда возникает вопрос, связанный с динамическим формированием списков полей. Рассмотрим подробнее наш пример, который демонстрирует, как создавать параметры для DoSortEx динамически. Предполагается, что пользователь сможет нажимать на заголовки DBGrid1, указывая, таким образом, поле, которое будет участвовать в сортировке. Повторное нажатие на колонку, которая уже участвует в сортировке, будет изменять порядок сортировки по этой колонке на противоположный. Сначала мы опишем вспомогательный класс для хранения информации о полях, которые будут участвовать в сортировках.

```
type

  TOrderStringList = class(TStringList)
  protected
    function GetAscending(Index: Integer): boolean;
    procedure SetAscending (Index: Integer; Value: boolean);
  public
    property Ascending[Index: Integer]: boolean read GetAscend-
ing write SetAscending;
  end;

…

{ TOrderStringList }

function TOrderStringList.GetAscending(Index: Integer): boole-
an;
begin
  Result := boolean(integer(Objects[Index]));
end;

procedure TOrderStringList.SetAscending(Index: Integer; Value:
boolean);
begin
  Objects[Index] := pointer(integer(Value));
end;
```

Очевидно, что данный класс является списком строк – названия полей, а также хранит для каждого поля, включенного в список, порядок сортировки в виде свойства Ascending. Ниже вы видите описание класса формы, а также два основных обработчика событий (OnCreate, OnDestroy). При создании формы мы создаем экземпляр класса TOrderStringList, а при ее уничтожении – удаляем.

```
  TMainForm = class(TForm)
    MainDB: TpFIBDatabase;
    MainDS: TpFIBDataSet;
    MainTr: TpFIBTransaction;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    Button1: TButton;
    Label2: TLabel;
    procedure DBGrid1TitleClick(Column: TColumn);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    SortFields: TOrderStringList;
    procedure ReSort;
  end;
procedure TMainForm.FormCreate(Sender: TObject);
begin
  SortFields := TOrderStringList.Create;
```

```
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
  SortFields.Free;
end;
```

Теперь напишем обработчик события OnTitleClick у компонента DBGrid1:

```
procedure TMainForm.DBGrid1TitleClick(Column: TColumn);
const OrderStr: array [boolean] of string = ('(DESC)',
'(ASC)');
var aField: string;
    aFieldIndex: integer;
begin
  aField := Column.FieldName;
  aFieldIndex := SortFields.IndexOf(aField);

  if aFieldIndex = -1 then begin
    SortFields.Add(aField);
    SortFields.Ascending[SortFields.Count - 1] := true;

    Column.Field.DisplayLabel := Column.Field.FieldName + Or-
derStr[true];
  end
  else begin
    SortFields.Ascending[aFieldIndex] := not Sort-
Fields.Ascending[aFieldIndex];

    Column.Field.DisplayLabel := Column.Field.FieldName +
       OrderStr[SortFields.Ascending[aFieldIndex]];
  end;
  ReSort;
end;
```

Смысл обработчика состоит в следующем: при нажатии пользователем на заголовок мы проверяем, есть ли данное поле в нашем списке сортировки. Если нет, то мы добавляем его и формируем новый заголовок для колонки, который теперь будет состоять из названия поля и порядка сортировки (ASC) или (DESC). Если же поле уже было включено в сортировку, то мы лишь меняем порядок сортировки. В обоих случаях мы должны вызвать процедуру ReSort, описанную ниже:

```
procedure TMainForm.ReSort;
var Orders: array of boolean;
    Index: Integer;
begin
  if SortFields.Count = 0 then begin
    MainDS.CloseOpen(false);
    exit;
  end;

  SetLength(Orders, SortFields.Count);

  for Index := 0 to pred(SortFields.Count) do
```

```
      Orders[Index] := SortFields.Ascending[Index];

  MainDS.DoSortEx(SortFields, Orders);
end;
```

Данная процедура формирует списки для метода DoSortEx на основе списка SortFields и пересортировывает записи. В случае, если наш список сортировки пустой, мы должны вернуться к "стандартному" порядку записей. Для этого вызывается метод CloseOpen.

> Параметр False означает, что мы не хотим автоматически получать сразу все записи от сервера.

Если наш список полей не пустой, то мы должны сформировать массив Orders. Это делается, как видно, достаточно легко. Используя динамические массивы, мы сначала задаем длину Orders равной количеству полей в списке SortFields, а потом последовательно заполняем Orders значениями свойства Ascending списка SortFields. Остается только вызвать метод DoSortEx – и сортировка будет выполнена.

Чтобы закрыть вопрос целиком, нам остается только предоставить пользователю возможность исключать поля из сортировки. Для этого мы положим на форму кнопку Button1 (см. рис. 46, заголовок "Delete column from Sorting"):

```
procedure TMainForm.Button1Click(Sender: TObject);
var aField: string;
    aFieldIndex: integer;
begin
  aField := DBGrid1.SelectedField.FieldName;
  aFieldIndex := SortFields.IndexOf(aField);

  if aFieldIndex <> -1 then begin
    SortFields.Delete(aFieldIndex);
    DBGrid1.SelectedField.DisplayLabel := aField;
    ReSort;
  end;
end;
```

Пользователь выделяет поле, которое хочет удалить из сортировки, и нажимает на кнопку "Delete column from Sorting", после чего это поле удаляется из списка SortFields и производится пересортировка записей.

Локальная фильтрация

Аналогично локальной сортировке, которая оперирует только с данными в локальном буфере, мы можем также выбирать записи из уже полученных по какому-либо критерию, скрывая от пользователя "лишние" записи.

Рассмотрим локальную фильтрацию на примере приложения Filtering. Этот пример включен в стандартную поставку FIBPlus. В примере используется база данных FIBPlus_Example.gdb (рис. 2.68).

> Эта база данных в виде backup-файла доступна на сайте http://www.fibplus.net/

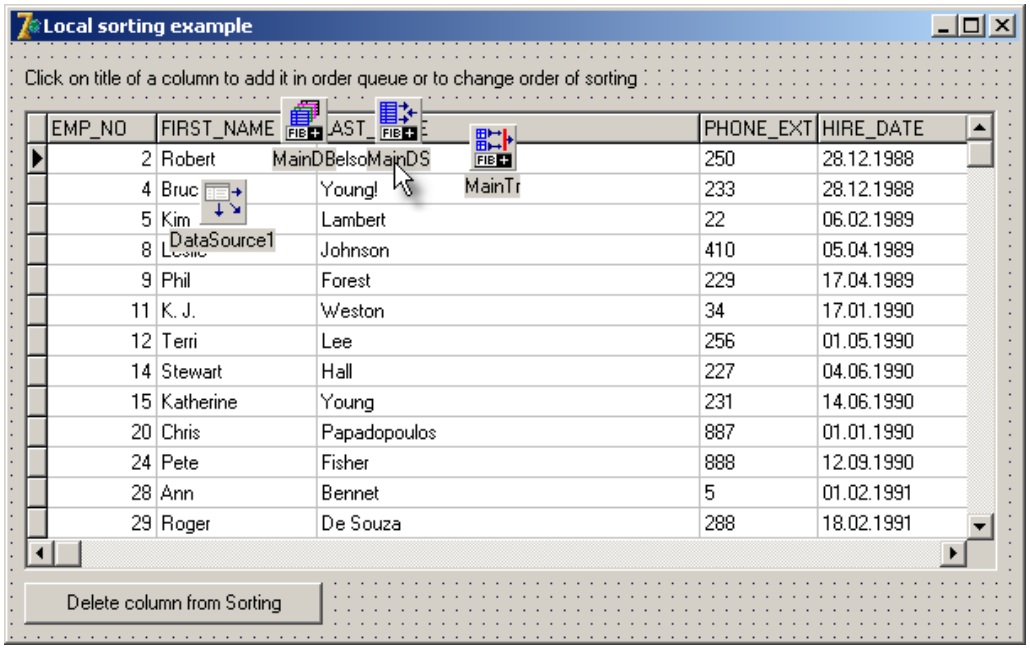


Рис. 2.68. Использование локальной фильтрации TpFIBDataSet

Совершенно очевиден код для подключения к базе данных. Список элементов компонента FieldsC: TComboBox заполняется названиями полей из таблицы BIOLIFE:

```
FilteringDS.SelectSQL: SELECT * FROM BIOLIFE

procedure TMainForm.btnConnectClick(Sender: TObject);
var Index: Integer;
begin
  with MainDB do begin
    ConnectParams.UserName := edtUserName.Text;
    ConnectParams.Password := edtPassword.Text;
    DBName := edtDataBase.Text;
    try
      Open;
      FilteredDS.Open;
      FieldsC.Items.Clear;
      for Index := 0 to pred(FilteredDS.FieldCount) do
        FieldsC.Items.Add(FilteredDS.Fields[Index].FieldName);
    except
      MessageDlg('Error of connection to a database!', mtError,
[mbOk], 0);
      Close;
    end;
  end;
end;
```

Пользователь может выбрать поле из списка FieldsC, указать в поле FilterE: TEdit строку для поиска и после нажатия на кнопку Button1 ("Activate Filter"):

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  FilteredDS.Filtered := false;
  FilteredDS.Filtered := true;
end;
```

в DBGrid1 останутся видны только те записи, которые содержат в заданном поле искомую строку. Для этого нам необходимо написать обработчик события OnFilterRecord у FilteringDS:

```
procedure TMainForm.FilteredDSFilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
    Accept := pos(FilterE.Text, Fil-
teredDS.FieldByName(FieldsC.Text).AsString) <> 0
end;
```

После включения FilteredDS.Filtered := true OnFilterRecord вызывается для каждой записи в локальном буфере. Если мы хотим, чтобы конкретная запись оставалась видимой, мы должны задать для нее параметр Accept равным True. Из примера кода видно, что мы оставляем видными только те записи, которые содержат в искомом поле (FieldsC.Text) заданную строку (FilterE.Text). Если мы, например, выберем поле COMMON_NAME и укажем строку для поиска "A", то в результате получим только две видимые записи (рис. 2.69).

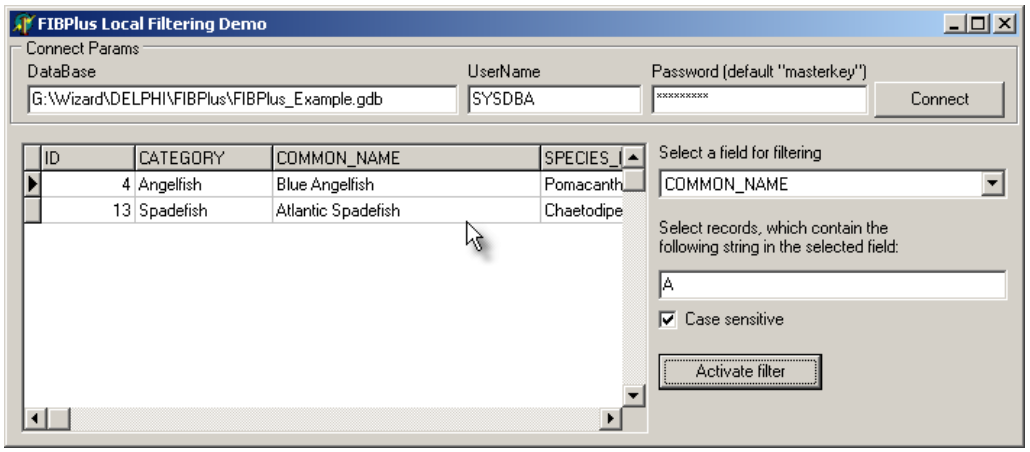


Рис. 2.69. Пример двух "отфильтрованных" записей

Поскольку наше условие для фильтра сформулировано таким образом, что мы выбираем записи по точному включению искомой подстроки – в нашем случае это только те записи, которые содержат большую букву "A". Мы можем немного усложнить условие поиска. Положим на форму CaseC: TcheckBox (см. рис. 2.69, "Case sensitive") и перепишем обработчик OnFilterRecord следующим образом:

```
procedure TMainForm.FilteredDSFilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
```

```
begin
  if CaseC.Checked then
    Accept := pos(FilterE.Text, Fil-
teredDS.FieldByName(FieldsC.Text).AsString) <> 0
  else
    Accept := pos(AnsiUpperCase(FilterE.Text),
                  AnsiUpper-
Case(FilteredDS.FieldByName(FieldsC.Text).AsString)) <> 0;
end;
```

Очевидно, что теперь если пользователь отключит "флажок" "Case sensitive", то фильтр по строке "A" будет содержать гораздо большее количество записей – все записи, которые содержат хотя бы одну букву "A", неважно маленькую или большую (рис. 2.70).

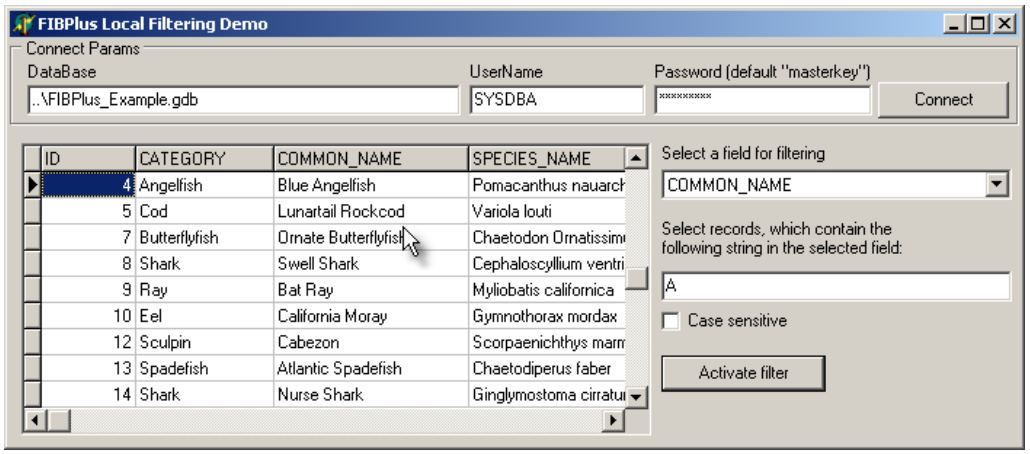


Рис. 2.70. Записи, содержащие букву "A"

Очевидно, что использование локальной фильтрации дает разработчику в руки очень гибкое средство построения условий, поскольку этот механизм не столько ограничен, как SQL в условиях WHERE. На основе локальной фильтрации, например, очень легко создается механизм быстрого поиска записи по первым буквам, которые пользователь набирает в поле редактирования.

Обработка событий InterBase при помощи FIBPlus

InterBase дает разработчику некоторый механизм для синхронизации приложений в многопользовательской среде Event Alerts. Суть данного механизма состоит в том, что вы можете вызывать пользовательские события из триггеров или хранимых процедур при помощи функции POST_EVENT:

```
CREATE PROCEDURE SHOW_EVENT(
EVENT_ID INTEGER)
AS
BEGIN
  IF (:EVENT_ID = 1) THEN POST_EVENT 'TEST_EVENT1';
  IF (:EVENT_ID = 2) THEN POST_EVENT 'TEST_EVENT2';
```

```
  IF (:EVENT_ID = 3) THEN POST_EVENT 'TEST_EVENT3';
  EXIT;
END
```

Создадим приложение (рис. 2.71), которое будет вызывать данную процедуру и получать соответствующие события. Если вы впоследствии попробуете запустить это приложение на двух компьютерах, то оба приложения будут получать сообщения вне зависимости от того, кто их инициировал.

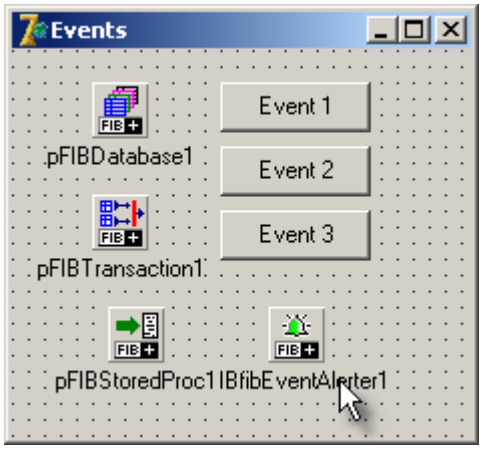


Рис. 2.71. Использование InterBase events, при помощи компонента TSIBfibEventAlerter

Для вызова хранимой процедуры мы будем использовать специальный компонент pFIBStoredProc: TpFIBStoredProc, задав свойство StoredProcName (рис. 2.72).

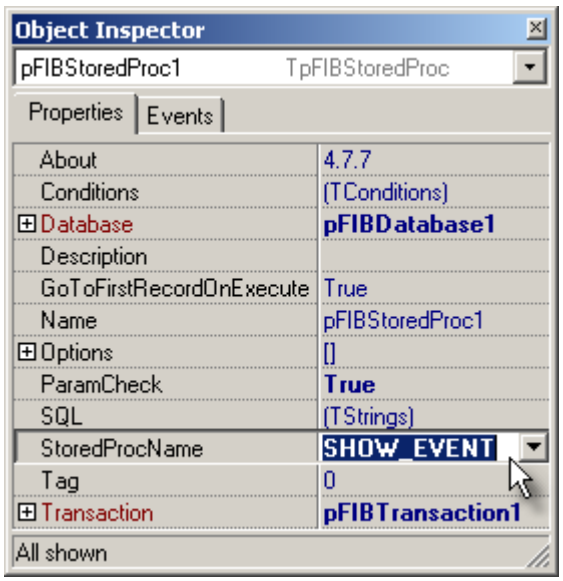


Рис. 2.72. Вызов хранимой процедуры при помощи компонента TpFIBStoredProc

Компонент TpFIBStoredProc является прямым потомком TpFIBQuery и сам формирует свойство SQL в виде 'EXECUTE PROCEDURE …'. Название процедуры задается свойством StoredProcName.

Компонент SIBfibEventAlerter1: TSIBfibEventAlerter1предназначен для получения приложением событий InterBase. Компонент ссылается на конкретное подключение к базе данных, т. е. на компонент типа TpFIBDatabase (рис. 2.73).

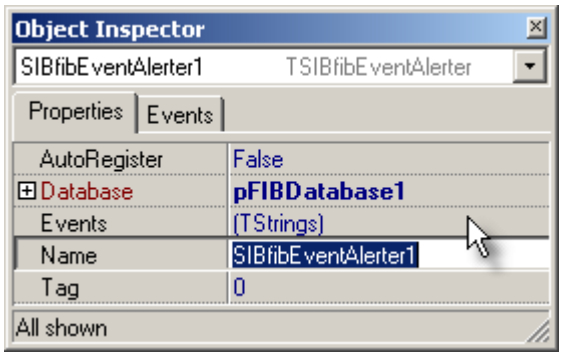


Рис. 2.73. Свойства SIBfibEventAlerter1

Установим свойство AutoRegister в False, чтобы собственноручно зарегистрировать нужные нам события, а в свойстве Events перечислим те события, которые нас интересуют (рис. 2.74).

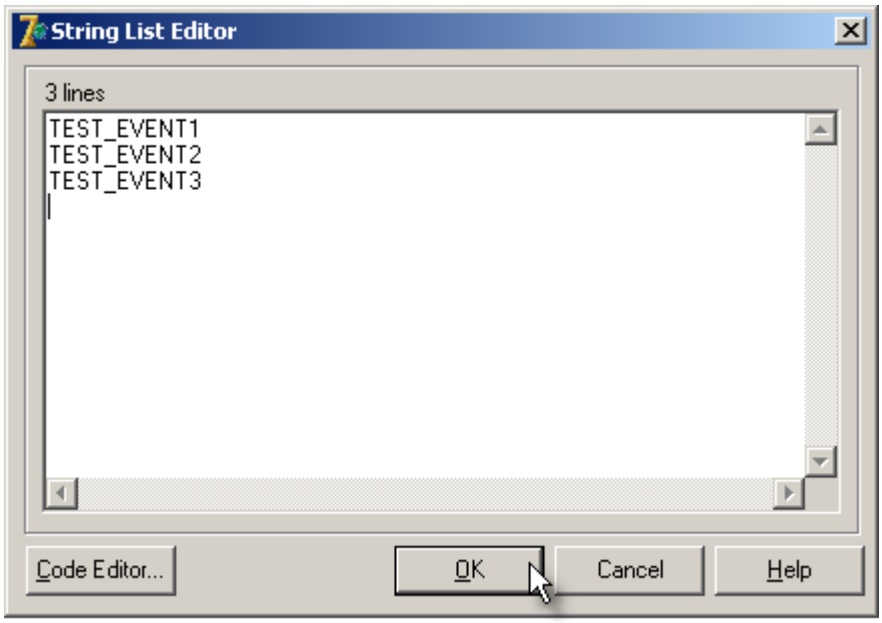


Рис. 2.74. Список зарегистрированных событий

Чтобы компонент SIBfibEventAlerter1 "получал" события, мы должны зарегистрировать их после подключения к базе данных. Для этого напишем обработчик события OnConnect у компонента pFIBDatabase1:

```
procedure TForm1.pFIBDatabase1Connect(Sender: TObject);
begin
  if not SIBfibEventAlerter1.Registered then
    SIBfibEventAlerter1.RegisterEvents;
end;
```

Соответственно перед закрытием подключения желательно "отключить" SIBfibEventAlerter1. Для этого создадим обработчик события BeforeDisconnect компонента pFIBDatabase1:

```
procedure TForm1.pFIBDatabase1BeforeDisconnect(Sender:
TObject);
begin
  if SIBfibEventAlerter1.Registered then
    SIBfibEventAlerter1.UnRegisterEvents;
end;
```

Зададим свойство Tag у кнопок равным соответственно 1, 2 и 3, и напишем общий обработчик события OnClick для них:

```
procedure TForm1.ButtonsClick(Sender: TObject);
begin
  if not pFIBTransaction1.InTransaction then
     pFIBTransaction1.StartTransaction;
  pFIBStoredProc1.Params[0].AsInteger := TButton(Sender).Tag;
  try
    pFIBStoredProc1.ExecProc;
    pFIBTransaction1.Commit;
  except
    pFIBTransaction1.Rollback;
  end;
end;
```

Смысл кода очевиден – мы задаем значение параметра процедуры, используя свойство Tag у кнопки, на которую нажал пользователь, выполняем процедуру при помощи метода ExecProc и либо подтверждаем транзакцию, либо отменяем ее в случае ошибки. Нам остается только написать обработчик события OnEventAlert компонента SIBfibEventAlerter1:

```
procedure TForm1.SIBfibEventAlerter1EventAlert(Sender: TObject;
  EventName: String; EventCount: Integer);
begin
  ShowMessage(EventName + ' : ' + IntToStr(EventCount));
end;
```

Параметр EventName возвращает название произошедшего события, а EventCount возвращает количество событий EventName. Значение параметра EventCount требует некоторых разъяснений. Итак, при работе с событиями важно помнить, что реально они вызываются только в случае подтверждения транзакции. Иными словами, если вы вызываете некоторое событие 'NEW_CUSTOMER'

при вставке записей в таблицу, и вызываете Commit только после вставки 1000 записей, то обработчик OnEventAlert будет вызван только один раз, сразу после Commit, и значение параметра EventCount будет равно 1000. Очевидно, что событие произошло 1000 раз, но приложение получает нотификацию обо всех событиях только после подтверждения транзакции, в контексте которой они происходили.

Мы можем запустить наше приложение. При нажатии на любую из кнопок мы будем получать сообщение о произошедшем событии, причем, как уже было сказано, если наше приложение запущено несколько раз и даже на разных компьютерах, все копии приложения будут получать нотификацию о событиях.

"Низкоуровневая" работа с внутренним буфером TpFIBDataSet

TpFIBDataSet включает несколько специальных методов для работы с внутренним буфером, в котором хранятся записи. В общем-то, данные методы превращают TpFIBDataSet в аналог TClientDataSet, ориентированный на InterBase. Наиболее интересным примером использования данных методов будет, пожалуй, построение диалога выбора, состоящего из двух списков (рис. 2.75).

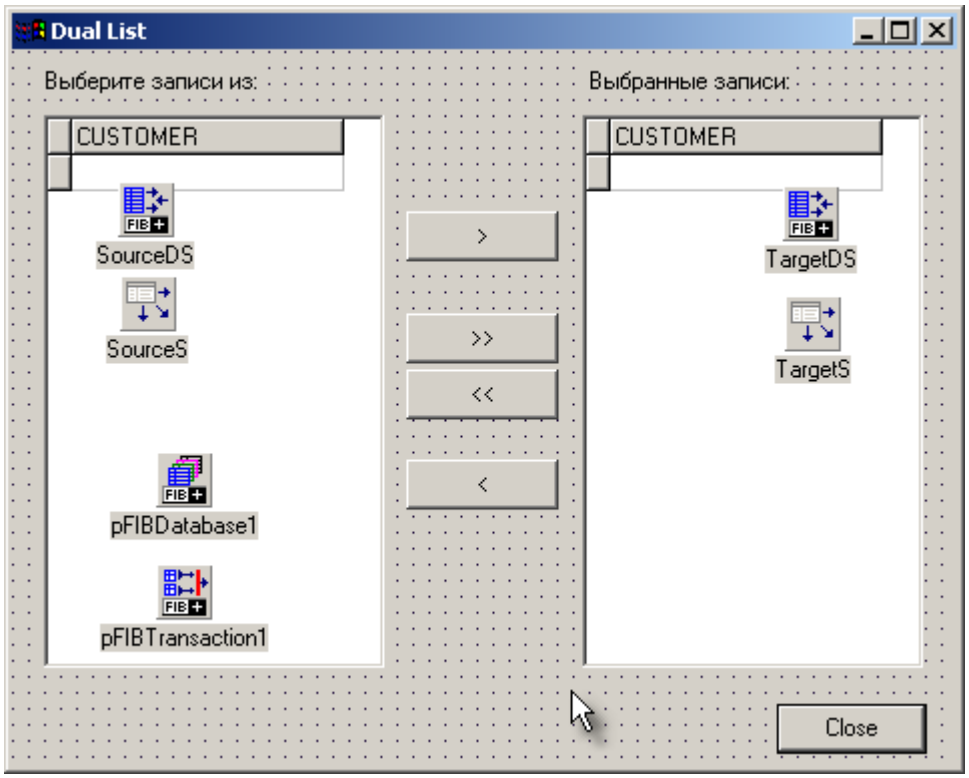


Рис. 2.75. Использование локального кеша TpFIBDataSet для организации двойного списка выбора

В данном примере мы будем использовать базу данных EMPLOYEE.GDB, входящую в стандартную поставку InterBase и FIBPlus. Покажем запросы для SourceDS:

SelectSQL:

```
SELECT
    CUS.CUST_NO,
    CUS.CUSTOMER
FROM
    CUSTOMER CUS
ORDER BY CUS.CUSTOMER
```

UpdateSQL:

```
UPDATE CUSTOMER SET
    CUSTOMER = ?CUSTOMER
 WHERE
            CUST_NO = ?OLD_CUST_NO
```

InsertSQL:

```
INSERT INTO CUSTOMER(
    CUST_NO,
    CUSTOMER
)
VALUES(
    ?CUST_NO,
    ?CUSTOMER
)
```

DeleteSQL:

```
DELETE FROM CUSTOMER
WHERE
            CUST_NO = ?OLD_CUST_NO
```

RefreshSQL:

```
SELECT
    CUS.CUST_NO,
    CUS.CUSTOMER
FROM
    CUSTOMER CUS
 WHERE
    (
            CUS.CUST_NO = ?OLD_CUST_NO
    )
```

Те же самые запросы мы будем использовать в TargetDS, поскольку оба компонента должны обладать совместимой структурой полей. Откроем оба запроса сразу после создания формы:

```
procedure TDualListForm.FormCreate(Sender: TObject);
begin
  SourceDS.Open;
  TargetDS.CacheOpen;
end;
```

> Если мы не указываем в коде явного открытия базы данных, то это означает, что свойство Connected у компонента TpFIBDatabase было задано в True в design-time.

Обратите внимание на то, что TargetDS мы активируем при помощи специального метода CacheOpen. Этот метод не выполняет запрос из SelectSQL, а только подготавливает внутренний буфер компонента в зависимости от запроса в SelectSQL. Очевидно, что после запуска приложения мы увидим записи в левом списке и пустую таблицу в правом. Теперь мы можем написать процедуру, которая позволит переносить записи из одного списка в другой:

```
procedure TDualListForm.MoveRec(Select: Boolean);
begin
  if Select then begin
    if (SourceDS.Active) and (not SourceDS.IsEmpty) then begin
      TargetDS.CacheRefreshByArrMap(SourceDS, frkInsert,
['CUST_NO'], ['CUST_NO']);
      SourceDS.CacheDelete;
    end;
  end
  else begin
    if (TargetDS.Active) and (not TargetDS.IsEmpty) then begin
      SourceDS.CacheRefresh(TargetDS, frkInsert, nil);
      TargetDS.CacheDelete;
    end;
  end;
end;
```

Если параметр Select равен True, то мы будем переносить текущую запись из левого списка в правый, т. е. из SourceDS в TargetDS. Рассмотрим подробнее метод CacheRefreshByArrMap.

```
procedure CacheRefreshByArrMap(FromDataSet: TDataSet; Kind:
TCachRefreshKind; const SourceFields, DestFields: array of
String);
```

Метод позволяет вставлять или модифицировать текущую запись в локальный буфер компонента, т. е. без выполнения соответствующих модифицирующих запросов. Параметр FromDataSet указывает, из какого TpFIBDataSet мы хотим получить запись. Параметр Kind может принимать два значения: frkEdit или frkInsert. FrkInsert указывает, что мы хотим вставить новую запись, а frkEdit – что мы хотим изменить существующую. Параметр SourceFields описывает список полей из компонента источника, а список DestFields – список полей в TpFIBDataSet-приемнике. Формально названия полей в списках могут не совпадать, но необходимо, чтобы совпадали типы данных, иначе обмен данными будет невозможен. В нашем примере, как вы можете видеть, мы вставляем в TargetDS текущую запись из SourceDS. Данный метод использован здесь только для демонстрации, поскольку мы вполне могли бы обойтись методом CacheRefresh, который копирует соответствующие поля автоматически. После вставки записи необходимо удалить ее из компонента источника. Это делается при помощи метода CacheDelete, который также не выполняет соответствующий запрос (DeleteSQL), а удаляет запись только из локального буфера в памяти. Теперь, когда мы знаем, что делают процедуры CacheRefreshByArrMap, CacheRefresh и CacheDelete, код процедуры достаточно

очевиден: мы копируем запись из компонента-источника в компонент-приемник и удаляем ее из компонента-источника.

То же самое можно сделать в цикле, если мы хотим перенести все записи сразу:

```
procedure TDualListForm.MoveAll(Select: Boolean);
var TmpDataSet: TpFIBDataSet;
begin
  if Select then TmpDataSet := SourceDS else
  TmpDataSet := TargetDS;

  with TmpDataSet do begin
    try
      DisableControls;
      First;
      while not Eof do MoveRec(Select);
    finally
      EnableControls;
    end;
  end;
end;
```

Остается написать обработчики события OnClick для кнопок:

```
procedure TDualListForm.Button1Click(Sender: TObject);
begin
  MoveRec(True); // SourceDS -> TargetDS
end;

procedure TDualListForm.Button2Click(Sender: TObject);
begin
  MoveRec(False); // TargetDS -> SourceDS
end;

procedure TDualListForm.Button3Click(Sender: TObject);
begin
  MoveAll(True); // SourceDS -> TargetDS
end;

procedure TDualListForm.Button4Click(Sender: TObject);
begin
  MoveAll(False); // TargetDS -> SourceDS
end;
```

Мы можем запустить наше приложение (рис. 2.76).

Если мы будем нажимать на кнопку Button1, то записи будут переноситься из левого списка в правый (рис. 2.77).

Все операции осуществляются только с данными внутри локальных буферов, не затрагивают сервер и реальные данные в базе. Наряду с описанными выше методами вы также можете использовать:

OpenAsClone – открыть DataSet и скопировать данные из другого DataSet.

CacheAppend – добавить запись в конец локального буфера.

SwapRecords – обменять записи в локальном буфере.

CacheEdit – задать значения полей у записи в локальном буфере.

CacheInsert – вставить запись в локальный буфер.

CacheModify – общий метод для вставки и изменения записи в локальным буфере.

LoadFromFile – заполнить локальный буфер записями из файла.

LoadFromStream – заполнить локальный буфер записями из потока.

SaveToFile – сохранить локальный буфер в файл.

SaveToStream – сохранить локальный буфер в потоке.

Функции работы с локальным буфером предоставляют разработчику чрезвычайно гибкое средство манипуляции данными в клиентских приложениях, позволяя улучшать наглядность пользовательского интерфейса без выполнения дополнительных запросов к базе данных.

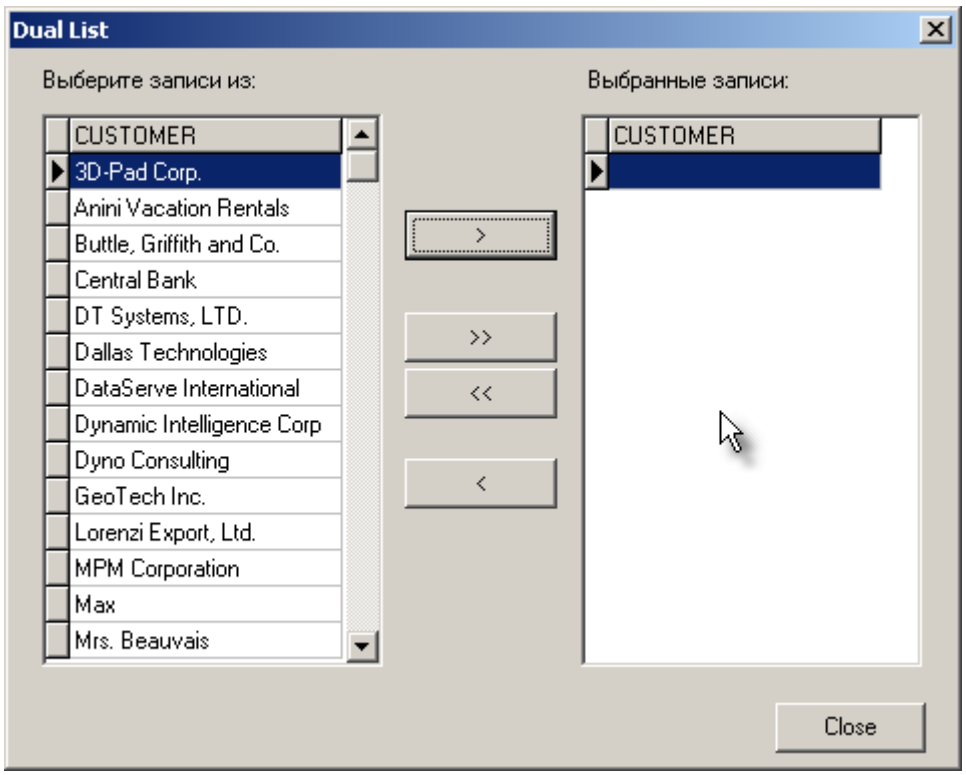


Рис. 2.76. Внешний вид запущенного приложения

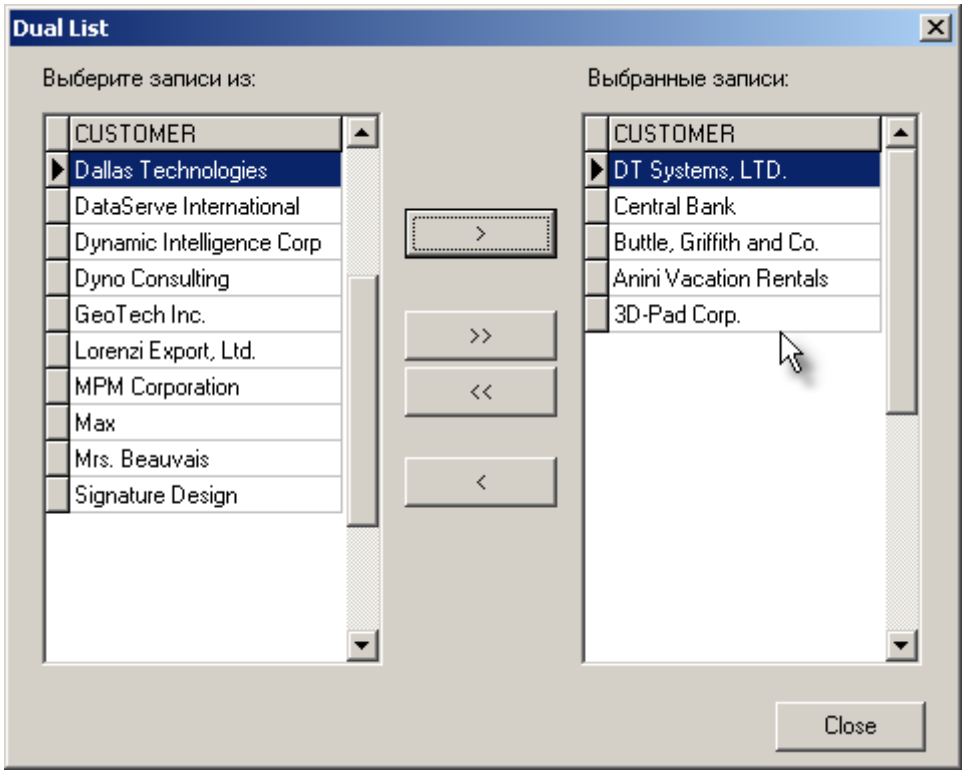


Рис. 2.77. Внешний вид приложения при выборе пользователем трех записей

Часть 3
Разработка приложений баз данных InterBase с использованием технологий Java, ODBC, CGI и Microsoft OLE DB

Разработка клиентских приложений СУБД InterBase с использованием технологии Microsoft OLE DB

(Материал для данной главы предоставлен Дмитрием Владимировичем Коваленко и публикуется с его любезного разрешения.)

Немного истории

Одним из распространенных заблуждений разработчиков баз данных является мысль, что СУБД InterBase ориентирована исключительно на работу с продуктами компании Borland. И этому способствовало то, что до последнего времени все качественные библиотеки доступа к этому серверу баз данных существовали только для создания приложений на Delphi, C++ Builder или Kylix. Для остальных систем программирования приходилось использовать InterBase API или ODBC. И хотя первое позволяет создавать высокопроизводительные приложения, а второе обладает значительными претензиями на универсальность, оба подхода не в полной мере удовлетворяют требованиям современных программных систем, базирующихся на компонентных технологиях. Поэтому потребность в использовании компонентов доступа к InterBase, универсальных с точки зрения языка программирования, была. И вопрос их реализации заключался только в одном: что именно должны предоставлять эти компоненты и кто решится начать их разработку.

Ответ на первый вопрос дают принципы современной организации масштабируемой архитектуры программного обеспечения, подразумевающие использование компонентов и их группировку в раздельно компилируемые модули. Поэтому компоненты доступа должны обеспечить прозрачную интеграцию компонентов в пределах как одного модуля, так и нескольких. И технология Component Object Model (COM) позволяет без проблем применять эти принципы

на практике. Но полноценное использование этой технологии для создания крупных проектов с использованием InterBase осложнялось отсутствием готовой стандартизованной реализации COM-объектов доступа к этой СУБД. В результате разработчики программного обеспечения под InterBase вынуждены либо продолжать создавать монолитные приложения, либо самостоятельно решать проблемы совместного использования ресурсов СУБД малосвязанными между собой модулями программы. В первом случае осознанное ограничение возможностей программы экономит время. Во втором, опуская дополнительные трудозатраты на создание компонентов доступа, можно попасть в ловушку, которая в лучшем случае не позволяет сменить сервер базы данных, в худшем приводит к краху всего проекта. Как правило, разработчики это осознают, когда поздно что-либо менять.

Тем не менее попытки создания COM-объектов для доступа к InterBase были. Наиболее успешной попыткой можно считать библиотеку Visual Database Tools от компании Borland (VDBT). Это VCL-подобные компоненты для Visual Basic, работающие с InterBase через Borland Database Engine. Но библиотека VDBT была готовой реализацией COM-объектов доступа к InterBase, а не открытой спецификацией. Поэтому расширению и усовершенствованию не подлежала.

Спецификацию под названием OLE Database (OLE DB), предназначенную для создания компонентов доступа к базам данных, выпустила компания Microsoft, которая курирует и саму COM-технологию. Но Open Source InterBase 6 не имел собственного OLE DB-провайдера, поэтому ничего не оставалось, как начать самостоятельную разработку OLE DB for InterBase, известную ныне как *IBProvider*.

Обзор возможностей IBProvider

- Возможность работы со всей линейкой СУБД InterBase, начиная с версии 4.x и заканчивая клонами InterBase 6 – Firebird и Yaffil. Минимальным условием работы IBProvider является наличие на компьютере клиента динамической библиотеки GDS32.dll от InterBase 4 (см. главу "Состав модулей InterBase" (ч. 4)). IBProvider самостоятельно определяет уровень возможностей сервера (так называемый base level) и клиентской части (т. е. возможности GDS32.dll), а также диалект базы данных и автоматически подстраивается под эти параметры.
- Поддержка всех типов данных InterBase. Есть поддержка BLOB-полей (бинарных и текстовых), массивов и типов DECIMAL/NUMERIC (см. главу "Типы данных" (ч. 1)).
- Поддержка storage-объектов для работы с BLOB-полями. Эти объекты могут возвращаться клиенту и приниматься в качестве входящих параметров.
- Практически весь спектр OLE DB-типов. Помимо типов, непосредственно поддерживаемых InterBase, IBProvider способен принимать и возвращать беззнаковые целые числа, булевы значения, строки UNICODE и т. д.
- Встроенная поддержка конвертирования данных из одного типа в другой, преобразования массивов, бинарного и текстового представления BLOB-полей. Для преобразования типа данных NUMERIC используется библиотека для

работы с большими целыми числами, что обеспечивает естественную поддержку 64-битовых целых.

- Поддержка многопоточной работы. Компоненты провайдера самостоятельно обеспечивают синхронизацию доступа к своим ресурсам, поэтому клиент может не беспокоиться о проблемах параллельной работы с одним подключением к базе данных из нескольких потоков одного приложения.
- Отказоустойчивость. Для компонентов, работающих в составе серверных приложений, исключительно важна надежность. При разработке провайдера повсеместно используются мощные возможности языка Си++ для автоматического освобождения ресурсов и обработки исключительных ситуаций.
- Оптимизация работы с результирующим множеством SQL-запросов. В зависимости от требований клиента используется механизм либо однонаправленного доступа к выборке, либо произвольного. Для поддержки обработки большого количества данных автоматически применяются временные файлы, причем для доступа к ним используется 64-битовая адресация.
- Тридцатидвухбитовый кеш выбранных строк результирующего множества. Применение динамической системы приоритетов позволяет удерживать в заданном объеме памяти только наиболее часто используемые строки, а хеш-таблица обеспечивает эффективную навигацию по содержимому кеша. Таким образом, IBProvider способен с одинаковой производительностью обрабатывать как небольшие по размеру результирующие множества, так и очень большие, даже превышающие объем доступной оперативной памяти.
- Оптимизация работы с оперативной памятью. Во-первых, IBProvider использует две собственные "кучи" (heap) для динамического выделения памяти. Это снижает нагрузку на системную кучу. Во-вторых, IBProvider интенсивно запускает совместно используемые объекты, хранящие информацию только для чтения. Во время работы IBProvider создает глобальный пул (pool) объектов, что приводит к экономии памяти и позволяет уменьшить время создания и инициализации объектов и, таким образом, улучшить общую производительность приложения баз данных.
- Полная поддержка синтаксиса SQL. Также поддерживаются команды для создания/удаления базы данных и явного управления транзакциями.
- Работа с базой данных в режиме автоматического запуск и подтверждения транзакций (autocommit). По умолчанию этот режим выключен, так как он не является оптимальным для работы с InterBase, но при необходимости его можно включить.
- Полная поддержка параметризованных запросов. Можно использовать именованные и неименованные параметры, самостоятельно или автоматически формировать описания параметров и передавать их значения в обоих направлениях (in-out-параметры).
- Поддержка вызова хранимых процедур (сокращенно – ХП; подробнее о них см. главу "Хранимые процедуры" (ч. 1)). Провайдер распознает запросы вида "exec proc_name", "execute proc_name", "execute procedure proc_name" и возвращает результат работы хранимой процедуры через выходные (out) параметры.

Возможность получения метаданных из базы данных InterBase. Это списки таблиц, колонок, хранимых процедур, индексов, ограничений и т. д. (всего 26 видов метаданных). Помимо CASE-средств и систем построения отчетов, эта информация используется в Microsoft Distributed Query для выполнения *гетерогенных запросов* к нескольким базам данных под управлением различных (!) SQL-серверов (например, MS SQL) посредством OLE DB-провайдеров.

- Тщательное следование парадигмам объектно-ориентированного проектирования, а также двухлетнее тестирование в реальных системах СУБД гарантируют высокий уровень надежности и стабильности IBProvider, который идеально подходит для использования в составе программного обеспечения с круглосуточным режимом работы.

В настоящий момент, оставив позади большой объем работ по созданию OLE DB для InterBase, можно пересмотреть роль и назначение этого драйвера. Вытеснив оригинальную клиентскую часть GDS32.DLL на второй план, IBProvider предоставляет мощный объектно-ориентированный низкоуровневый клиентский API для работы с InterBase. Встраиваясь в приложения баз данных, OLE DB-провайдер способен взять на себя всю работу по организации взаимодействия с сервером базы данных. Предоставление ресурсов для работы с базой данных в виде COM-объектов снимает традиционные ограничения, накладываемые на клиентские приложения баз данных. Приложение можно дробить на модули, которые можно создавать с помощью разных систем программирования. Используя сценарии, написанные на VBScript/JScript, в программы можно добавлять логику, которую невозможно реализовать на уровне базы данных. OLE DB является общепризнанным промышленным стандартом доступа к данным, что позволяет легко разворачивать и управлять приложениями, разработанными с использованием IBProvider.

Таким образом, разработка крупных масштабируемых клиентских приложений для InterBase с помощью средств разработки компании Microsoft, а также любых других систем, поддерживающих OLE DB, становится более реальной и доступной, чем можно было себе представить ранее.

Использование IBProvider в клиентских приложениях

Низкоуровневые прикладные интерфейсы для работы с СУБД (API) обычно не используются в клиентских приложениях из-за большого объема кода, необходимого для подготовки и выполнения SQL-запросов. Это относится и к OLE DB-интерфейсам. Поэтому примеры, демонстрирующие взаимодействие с различными OLE DB-провайдерами непосредственно через их COM-интерфейсы, носят исключительно демонстрационный характер и имеют мало общего с реальным программированием приложений баз данных. Для решения повседневных задач обычно используют надстройки в виде других COM-объектов или библиотек классов, которые существенно упрощают работу с OLE DB. Работать с IBProvider можно двумя основными способами – через стандартные ADODB компоненты и с помощью собственной библиотеки классов для поддержки IBProvider, написанной для компилятора Borland C++ Builder.

Компоненты ADODB

В настоящее время этот набор компонентов стал промышленным стандартом взаимодействия с OLE DB-провайдерами. ADODB (www.microsoft.com/data) – это весьма удобный высокоуровневый интерфейс, реализующий классическую иерархию объектов для работы с базами данных в виде COM-объектов, поддерживающих технологию OLE Automation.

Реализация OLE DB-интерфейсов большинства провайдеров не является взаимозаменяемой и больше ориентирована на использование из программ, написанных на C++, что приводит к проблемам совместимости и переносимости приложений между различными OLE DB-провайдерами. Поэтому ADODB-компоненты, сглаживающие различия между разными OLE DB-провайдерами и доступные для использования практически везде – начиная от VisualBasic и заканчивая тем же C++, – лучше подходят для использования в качестве универсальной платформонезависимой основы для приложений баз данных.

Но у ADODB-компонентов имеется ряд недостатков, которые являются обратной стороной достоинств:

1. Ограничения на типы данных, накладываемые структурой VARIANT.
2. Некоторое снижение производительности за счет того, что создание объектов ADODB производится через инфраструктуру COM.
3. Отсутствие встроенной поддержки использования нескольких независимых транзакций в рамках одного подключения, чем InterBase выгодно отличается от других SQL-серверов базы данных.
4. Появляется дополнительный уровень взаимодействия с OLE DB.

Для того, чтобы избежать недостатков ADODB-компонентов, была разработана специализированная библиотека классов C++, которая реализует доступ к OLE DB провайдерам (в том числе и к IBProvider) с максимально возможной эффективностью. Эта библиотека поставляется в составе дистрибутива IBProvider.

Библиотека классов C++ для работы с OLE DB

Созданная как дополнительный слой ("обертка") над COM-объектами, эта библиотека классов обеспечивает более тесную интеграцию с OLE DB-провайдерами. В ней нет всего спектра возможностей, который предлагает ADODB, но предоставляемый сервис делает ее более приспособленной для построения независимых и эффективных компонентов, работающих с базами данных через OLE DB.

К основным достоинствам данной библиотеки классов относится:

- автоматическое создание и разрушение объектов;
- изоляция классов для работы с базами данных друг от друга, что обеспечивает исключительную модульность и гибкость приложений на основе данной библиотеки;
- возможность подключения объекта C++ к уже существующему OLE DB компоненту;
- удобная работа с наборами полей и параметров, значительно более гибкая по сравнению с компонентами ADODB и VCL;

- возможность выбора способа обработки ошибок – через исключения или через код возврата.

Поскольку данная библиотека доступа создавалась специально для использования в больших проектах, её классы значительно уменьшают сложность взаимодействия с OLE DB-провайдером. В случае необходимости использовать ADODB (например, для совместной работы модулей проекта, написанных на C++ и на VBScript, в рамках одной транзакции) в библиотеке реализованы механизмы "шлюзования".

Разумеется, существуют еще несколько других библиотек, упрощающих работу с OLE DB-провайдерами. Однако в нижеследующих примерах будут использоваться только компоненты ADODB и библиотека классов C++ для работы с OLE DB. Поэтому, прежде чем приступить к работе над описанными примерами, убедитесь в наличии всех необходимых программных продуктов. Помните, что вы можете скачать все примеры и нужные для их работы программы на сайте поддержки данной книги www.InterBase-world.com.

Инсталляция IBProvider

Перед установкой OLE DB-провайдера убедитесь, что на вашей машине инсталлирована клиентская часть InterBase. Для этого на компьютере как минимум, должна находиться GDS32.DLL. Обычно она находится в системном каталоге Windows (System – для 95/98/ME, System32 – для NT4/Win2000). Подробнее об установке клиентской части InterBase см. главу "Установка InterBase – взгляд изнутри" (ч. 4).

В минимальный набор дистрибутива IBProvider входят два модуля: _IBProvider.dll и cw3250mt.dll. Скопируйте оба файла в системный каталог Windows и выполните команду regsvr32 _IBProvider.dll для регистрации провайдера в системе.

Если вы обладаете готовым дистрибутивом IBProvider, то программа инсталляции выполнит все необходимые операции самостоятельно.

Обратите внимание, что при инсталляции провайдера в Windows NT4/Windows 2000 у вас должны быть права на запись в реестр. Поэтому операцию регистрации лучше всего выполнять, обладая правами администратора.

После установки провайдера перезагрузка ОС не требуется.

Инсталляция ADODB-компонентов

Компоненты ADO входят в состав свободно распространяемого дистрибутива Microsoft Data Access Components и доступны для скачивания на сайте компании Microsoft – www.microsoft.com/data. Для написания примеров использовались ADODB-компоненты из дистрибутива версии 2.6.

Примеры использования ADODB

Для создания примеров работы с IBProvider через ADODB был применен Visual Basic for Application (VBA) из Microsoft Excel 97. Для использования ADODB-компонентов нужно их добавить в список библиотек, употребляемых Visual Basic. Для этого:

1. Откройте редактор кода Visual Basic (Alt+F11).

2. Выберите пункт меню ***Сервис\Ссылки***.
3. Найдите в списке строку Microsoft ActiveX Data Objects 2.6 Library и поставьте рядом с ней галочку.
4. Закройте окно, нажав кнопку "OK".

Использование библиотеки классов

Библиотека классов поставляется в виде исходных текстов. Поэтому для ее использования нужно выполнять следующие требования:

1. Явно добавить в проект файлы из каталога Lib:

ole_lib\oledb\oledb_client_lib.cpp	Основные классы для работы с OLE DB
ole_lib\oledb\oledb_client_base.cpp	
ole_lib\oledb\oledb_common.cpp	
ole_lib\oledb\oledb_variant.cpp	
ole_lib\oledb\oledb_ado_lib.cpp	Утилиты стыковки с ADODB
ole_lib\ole_base.cpp	
ole_lib\ole_auto.cpp	
Win32Lib\win32lib.cpp	
structure\util_classes.cpp	
util_func.cpp	

2. Начало каждого cpp-файла, включенного в проект, должно выглядеть следующим образом:

```
#include <_pch_.h>
#pragma hdrstop
```

3. Добавить в параметры проекта (опция Conditional defines) макрос INCLUDE_OLEDB_HEADER.
4. При использовании в проекте VCL компонент, нужно добавить в параметры проекта макрос _USE_VCL_. В этом случае файл <vcl.h> будет добавлен в проектный csm-файл (файл прекомпилированного заголовка) косвенно из <_pch_.h>.
5. Основной каталог include, используемый компилятором C++ Builder, должен содержать заголовочные файлы OLE DB SDK. BCB5 и Free Borland C++ Compiler уже содержат все необходимое. В BCB3 нужно добавить эти файлы самостоятельно, используя OLE DB SDK версии не выше 2.1.

Представленная в составе дистрибутива IBProvider библиотека классов является основой для проектов, её использующих. Поэтому предполагается, что заголовочный файл <_pch_.h> прямо или косвенно включен в каждый cpp-файл проекта. Возможность параллельного использования с другими библиотеками осуществляется за счет определения пространств имен. Поддержка библиотеки VCL добавлена изначально. Для поддержки других библиотек потребуется модифицировать <_pch_.h>.

Перенос на другие компиляторы C++ полностью зависит от степени их совместимости с последним стандартом C++ и от сложности перехода на другую реализацию STL.

Примеры использования библиотеки классов

Для написания и тестирования примеров использовался Borland C++ Builder 3-й версии (с установленным пакетом исправлений, который доступен для скачивания на сайте компании Borland). Библиотека классов самостоятельно конфигурируется под использование компилятора и STL из BCB5, поэтому примеры переносятся на Borland C++ Builder 5-й версии без проблем.

В примерах, включенных в текст этого раздела, опускаются этап инициализации COM и обработка исключений. Все это, естественно, присутствует в оригинале примеров, доступных для скачивания с сайта поддержки этой книги www.InterBase-world.com.

Также для изучения технологии использования библиотеки классов для работы с OLE DB из C++ рекомендуется посмотреть примеры из дистрибутива IBProvider.

Тестовая база данных

Для тестирования использовался Firebird 1.0 и база данных employee.gdb, входящая в дистрибутив этого сервера баз данных. На этом сервере был создан пользователь "gamer" с паролем "vermut".

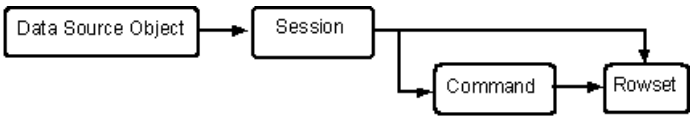
Для сокращения объема кода, проводящего подключение к базе данных, использовался текстовый файл employee.ibp, содержащий следующую информацию:

```
data source=c266:d:\database\employee.gdb;
user=gamer;
password=vermut;
auto_commit=true;
ctype=win1251;
```

Операционная система

Все перечисленные компоненты для написания примеров были установлены на одном компьютере, работающем под управлением Windows NT4 Service Pack 5, Internet Explorer 5.

Состав компонентов IBProvider



Давайте рассмотрим составные части IBProvider. Компоненты, входящие в состав OLE DB-провайдера, делятся на 4 основные группы:

Источник данных (Data Source). Компоненты этой группы отвечают за инициализацию и управление подключением к базе данных. Здесь же предоставляется интерфейс для создания сессии.

Сессия (Session). Компонент управления транзакцией. У одного источника данных может быть несколько сессий. Кроме управления транзакциями, сессия

обеспечивает создание команд, открытие таблиц и получение информации о метаданных базы данных.

Команда (Command). Компонент для подготовки и выполнения SQL-запросов к базе данных. Команда выполняется в рамках конкретной сессии, однако в случае работы в режиме автоматического запуска и подтверждения транзакций (autocommit) выполняется в рамках собственной транзакции. Внутри одной сессии может существовать множество команд.

Набор строк (Rowset). Компонент, реализующий интерфейсы навигации по результатам SQL-запроса (результирующему множеству) и доступа к его содержимому.

Для описания взаимодействующих сторон, будет использоваться следующая терминология:

Клиент – это любой фрагмент системного или прикладного кода, использующий интерфейс OLE DB. Сюда могут входить и сами компоненты доступа. Так же, для обозначения этой стороны взаимодействия, будут использоваться термины *пользователь* и *потребитель*.

Компонентов доступа – это любой программный компонент, предоставляющий интерфейс OLE DB. Параллельно будут использоваться *OLE DB-поставщик*, *OLE DB-провайдер, провайдер* и *IBProvider*.

Настройка и определение функциональности компонентов осуществляется через их свойства. *Свойства – это атрибуты объекта.* Например, свойства набора строк определяют верхний предел объема оперативной памяти для хранения данных, поддержку закладок и потоковую модель. Клиенты устанавливают значения свойств, чтобы потребовать от соответствующего объекта некоторого заданного поведения, и читают свойства, чтобы определить возможности объекта. Каждое свойство характеризуется значением, типом, описанием, атрибутом чтения/записи.

Свойство идентифицируется *GUID* (глобальный уникальный идентификатор, представляющий собой структуру длиной 128 бит) и целым числом, представляющим идентификатор свойства. *Набор свойств* – это совокупность свойств с одним и тем же GUID.

Источник данных

Создание компонента Data Source является отправной точкой для работы с базой данных через IBProvider. Существует несколько сценариев создания и инициализации компонента доступа. Они отличаются объемом работы, выполняемой в клиентском приложении, библиотекой доступа к OLE DB и самим IBProvider.

Вариант 1. Клиент самостоятельно осуществляет все этапы:

ADODB

```
Dim cn As New ADODB.Connection
cn.Provider = "LCPI.IBProvider.1"
cn.Properties("data source") =
"localhost:d:\database\employee.gdb"
cn.Properties("user id") = "gamer"
```

```
cn.Properties("password") = "vermut"
cn.Open
```

C++

```
t_db_data_source cn;
_THROW_OLEDB_FAILED(cn,create("LCPI.IBProvider.1"))
t_db_obj_props cn_props(/*refresh=*/false);
_THROW_OLEDB_FAILED(cn_props,attach_data_source(cn.m_obj,
DBPROPSET_DBINITALL))
_THROW_OLEDB_FAILED(cn_props,set("data source",
"localhost:d:\\database\\employee.gdb"));
_THROW_OLEDB_FAILED(cn_props,set("user id","gamer"));
_THROW_OLEDB_FAILED(cn_props,set("password","vermut"));
_THROW_OLEDB_FAILED(cn,attach(""));
```

Вариант 2. Создание и инициализацию выполняет клиентская библиотека:

ADODB

```
Dim cn As New ADODB.Connection
Call cn.Open("provider=LCPI.IBProvider.1; data
source=localhost:d:\database\employee.gdb", "gamer", "vermut")
```

C++

```
t_db_data_source cn;
_THROW_OLEDB_FAILED(cn,attach("provider=LCPI.IBProvider.1;"
           "data source=localhost:d:\\database\\employee.gdb;"
           "user id=gamer;password=vermut"));
```

Вариант 3. Провайдер создает клиентская библиотека, инициализацию выполняет сам провайдер:

ADODB

```
Dim cn As New ADODB.Connection
Call cn.Open("file name=d:\database\employee.ibp")
```

C++

```
t_db_data_source cn;
_THROW_OLEDB_FAILED(cn,
                  attach("file name=d:\\database\\employee.ibp"));
//или явно указываем провайдер и файл с параметрами
_THROW_OLEDB_FAILED(cn,
           attach("provider=LCPI.IBProvider.1;"
           "file name=d:\\database\\employee.ibp"));
```

где "employee.ibp" – обычный текстовый файл, в котором хранится строка подключения вида

```
data source=localhost:d:\\database\\employee.gdb;
user id=gamer;
password=vermut
```

Пока OLE DB-провайдер не подключен к базе данных, параметры инициализации будут единственно доступным набором свойств. В IBProvider определены стандартные свойства инициализации и собственные, предназначенные для спе-

циализированной настройки дальнейшей работы с базой данных. (За подробностями обращайтесь к документации по IBProvider.)

После успешного подключения к базе данных становятся доступными свойства информационного набора, с помощью которых компонент источника данных предоставляет сведения о сервере базы данных. Часть этих свойств стандартизовано спецификацией OLE DB. Остальные свойства предоставляют информацию, специфичную для IBProvider, содержащую расширенные сведения о сервере и базе данных.

Пример получения значений информационных свойств:

ADODB

```
'подключение к базе данных
….
'стандартные свойства
Debug.Print cn.Properties("provider version")
Debug.Print cn.Properties("provider friendly name")
'специфические свойства
Debug.Print cn.Properties("IB Base Level")
Debug.Print cn.Properties("IB GDS32 Version")
Debug.Print cn.Properties("IB Version")
```

C++

```
//Подключение к базе данных
//…
t_db_obj_props cn_props(false);
_THROW_OLEDB_FAILED(cn_props,
attach_data_source(cn.m_obj, DBPROPSET_DATASOURCEINFOALL));
//печать всех информационных свойств
for(UINT i=0;i!=cn_props.GetItemsInContainer();++i)
{

cout<<cn_props[i].name()<<":"<<print(cn_props[i].value())<<endl
;
}
```

Компонент Data Source имеет еще несколько возможностей: например, сохранение параметров подключения к базе данных в файле и перечисление допустимых символов для названий объектов базы данных. Однако в основном он используется для создания объектов сессий.

Сессия

Основная функция сессии – установить рамки транзакции с заданными параметрами (подробнее о транзакциях см. главу "Транзакции. Параметры транзакций" (ч. 1)).

Хотя в ADODB понятие сессии совмещено с понятием источника данных, в OLE DB это два различных объекта. Надо полагать, что основная причина такой иерархии объектов ADODB заключается в архитектуре пула подключений, используемого в серверных приложениях Microsoft. Гораздо проще и эффективнее осуществлять балансировку загрузки на уровне отдельных подключений к базе данных, чем на уровне сессий. Как правило, в SQL-серверах через два

раздельных подключения можно осуществлять параллельные запросы к базе данных, а через разные сессии одного подключения такая работа будет осуществляться последовательно. Тем не менее InterBase может одновременно обслуживать несколько транзакций в рамках одного подключения и в данном случае выгодно отличается от большинства других SQL-серверов. Поэтому IBProvider поддерживает возможность создания нескольких объектов сессий, принадлежащих одному источнику данных.

Уровни изоляции транзакции

В IBProvider реализована поддержка трех уровней изоляции транзакций: READ COMMITTED, REPEATABLE READ (SNAPSHOT), SERIALIZABLE (SNAPSHOT TABLE STABILITY). При работе через ADODB следует обратить внимание на значение свойства ADODB.Connection.IsolationLevel. Библиотека классов C++ по умолчанию использует режим REPEATABLE READ. Подробнее об уровнях изоляции транзакций вы можете узнать в главе "Транзакции. Параметры транзакций" (ч. 1).

Управление транзакциями

Поскольку корректное использование транзакций является важным условием разработки эффективных приложений баз данных, то IBProvider по умолчанию требует явного участия пользователя в процессе управления транзакциями.

Причинами отказа от автоматического запуска и завершения транзакции являются относительно высокие затраты ресурсов на эту операцию и желание контролировать все действия с базой данных.

Надо сказать, что последнее обстоятельство является наиболее важным, поскольку при разработке большой программной системы, состоящей из множества независимых модулей, явный контроль операций с базой данных позволяет избегать многих ошибок.

В то же время иногда автоматический запуски завершение транзакции могут оказаться очень удобными для решения небольших и несложных задач.

Помимо автоматических транзакций для работы с данными в IBProvider определена еще одна категория *внутренних* транзакций (inner transaction), используемых для чтения метаданных базы данных. По умолчанию *внутренние* транзакции разрешены, поскольку CASE-дизайнеры и системы построения отчетов для получения метаданных явно не управляют транзакциями.

Автоматические транзакции

Для разрешения провайдеру самостоятельно управлять транзакциями нужно указать в строке инициализации параметр "auto_commit=true":

```
Call cn.Open(
    "data source=localhost:d:\database\employee.gdb;auto_commit=true",
    "gamer", "vermut")
```

В этом случае все создаваемые объекты сессий для данного источника данных будут способны самостоятельно запускаться и завершать транзакции без явного участия пользователя. Для выборочного разрешения такого режима можно воспользоваться свойством сессии "Session AutoCommit". Если это свойство равно true, то сессия может обслуживать запросы к базе данных без необходимости явного запуска и подтверждения.

По умолчанию автоматические транзакции используют уровень изоляции SNAPSHOT. Если требуется установить другой уровень изоляции, то нужно либо установить параметр инициализации источника данных "auto_commit_level", либо изменить свойство сессии "Autocommit Isolation Levels" в одно из следующих значений:

- 0x1000 – READ COMMITED;
- 0x10000 – REPEATABLE READ. Это режим по умолчанию;
- 0x100000 – SERIALIZABLE.

О принципе функционирования автоматических транзакций следует сказать следующее. Автоматическая транзакция не управляется через интерфейс сессии. За её завершение и откат отвечает команда или набор строк внутри этой сессии. Если команда не возвращает набор строк (rowset), то транзакция фиксируется сразу. Если внутри автоматической транзакции появляется набор строк, то транзакция завершается при освобождении этого набора. То есть внутри одной сессии, у которой свойство "Session AutoCommit" равно true, может существовать несколько активных транзакций. Естественно, что в этом случае пользователь также может явно управлять запуском и завершением транзакции, принадлежащей сессии. В случае явного запуска транзакции, для дальнейших запросов и операций в рамках этой сессии будет использоваться контекст именно этой транзакции.

Несмотря на то, что IBProvider старается выполнить как можно больше операций в рамках одной автоматической транзакции, все равно производительность приложения, не осуществляющего самостоятельного контроля над транзакциями, не может сравниться с приложениями, явно управляющими транзакциями.

Управление транзакциями через SQL

Помимо управления транзакцией через OLE DB-интерфейсы сессии, IBProvider осуществляет специальную поддержку SQL-запросов вида: "SET TRANSACTION…", "COMMIT" и "ROLLBACK". В этом случае будет использоваться транзакция, принадлежащая сессии. Управление транзакциями через SQL-запросы позволяет указывать специфические параметры контекста транзакции, которые не стандартизированы в OLE DB и которые возможно использовать благодаря особенностям InterBase API.

Примеры работы с транзакциями

Ниже приведены примеры, демонстрирующие способы запуска и завершения транзакций, поддерживаемые OLE DB-провайдером.

ADODB:

явное управление транзакцией:

```
Dim cn As New ADODB.Connection
cn.Provider = "LCPI.IBProvider.1"
Call cn.Open(
"data source=localhost:d:\database\employee.gdb",
"gamer","vermut")
'стандартный способ
cn.IsolationLevel = adXactRepeatableRead
cn.BeginTrans
'...
'можно указать использование commit retaining
'cn.Attributes = adXactAbortRetaining + adXactCommitRetaining
cn.CommitTrans
'управление транзакцией через SQL (можно использовать специфику
InterBase)
Dim cmd As New ADODB.Command
cmd.ActiveConnection = cn
cmd.CommandText = "set transaction"
cmd.Execute
'...
cmd.CommandText = "rollback"
cmd.Execute
```

автоматический запуск транзакций – разрешение и запрещение:

```
Dim cn As New ADODB.Connection
cn.Provider = "LCPI.IBProvider.1"
'auto_commit=true - включение автоматических транзакций
' для всех сессий
Call cn.Open(
"data source=localhost:d:\database\employee.gdb;auto_commit=true",
"gamer", "vermut")
Dim cmd As New ADODB.Command
Dim rs As ADODB.Recordset
cmd.ActiveConnection = cn
cmd.CommandText = "select * from rdb$database"
Set rs = cmd.Execute 'транзакция запускается неявно
'...
'и будет завершена при закрытии результирующего множества
rs.Close
'альтернативой auto_commit=true является установка
'(после успешного подключения к база данных)
'свойства сессии Session AutoCommit=true.
'В ADODB это одно и то же,
'поскольку на одно подключение - одна сессия
'через это же свойство можно "выключить" глобальное
'разрешение на автоматический
'запуск транзакции, что дальше и демонстрируется
cn.Properties("Session AutoCommit") = False
cmd.CommandText = "select * from rdb$database"
Set rs = cmd.Execute
'выдаст ошибку "Automatic transaction is disabled"
```

При программировании на C++ принципы взаимодействия с сессией точно такие же. Но в C++ сессия будет отдельным объектом.

В нижеследующем примере на C++ демонстрируется трюк, который часто используется для моделирования принудительного завершения транзакций. Дело в том, что InterBase реализует многоверсионную архитектуру данных (подробнее о многоверсионности данных см. главу "Транзакции. Параметры транзакций" (ч. 1)). Одной из особенностей такой архитектуры является то, что любые транзакции желательно завершать подтверждением (commit), а не откатом (rollback).

Однако, как правило, транзакцию, в рамках которой выполняется операция чтения данных, не подтверждают, а откатывают. Особенно в случае достаточно длинного участка кода, который генерирует исключения в случае непредвиденных ошибок загрузки информации. Поэтому и нужно принудительно завершать транзакции так, как это показано в нижеследующем примере.

```
try
{
 t_db_data_source cn;
 _THROW_OLEDB_FAILED(cn,attach("provider=LCPI.IBProvider.1;"
         "data source=localhost:d:\\database\\employee.gdb;"
         "user id=gamer;"
         "password=vermut"));
 t_db_session session;
 // метод create перегружен для разных типов аргумента, поэтому
 // можно передавать
 // как C++ объект (t_db_data_source), так и IUnknown источника
 // данных
 _THROW_OLEDB_FAILED(session,create(cn));
 // запуск транзакции
 _THROW_OLEDB_FAILED(session,start_transaction());
 // создаем объект для принудительного завершения транзакции
 t_auto_mem_fun_1<HRESULT,bool,t_db_session>
  _auto_commit_(session, /*commit_retaining=*/false,
&t_db_session::commit);
 //... теперь, что бы ни произошло, транзакция
 // будет "закоммичена"
 throw runtime_error("This is my test error");
}
catch(const exception& exc)
{
 cout<<"error:"<<exc.what()<<endl;
}
```

По умолчанию библиотека не генерирует исключений, поэтому даже если сбой произошел из-за проблем с самой базой данных, то принудительное завершение транзакции, выполняемое в деструкторе _auto_commit_, не возбудит исключения.

Распределенные транзакции

Еще одним способом инициирования транзакции является подключение сессии к *координатору распределенных транзакций*. В общих чертах, координатор представляет собой сессию, транслирующую вызовы собственных интерфейсов управления транзакциией в идентичные групповые вызовы интерфейсов списка дочерних сессий. Как правило, от пользователя не требуется никакого участия для подключения к координатору. За это отвечает окружение, у которого пользовательский код запрашивает подключение к базе данных.

На практике распределенные транзакции обычно используются в COM+ (MTS) и Microsoft Distributed Query. В первом случае существует следующая особенность. Когда компоненты просят у своего окружения предоставить им подключение к базе данных, то на каждый такой запрос создается отдельная пара источник данных – сессия. Связано это с описанным выше принципом работы пула подключений. Если компонентов, обрабатывающих пользовательский запрос, очень много и они работают с одной и той же базой данных с идентичными параметрами подключения, то имеет смысл один раз получить подключение и потом предоставлять его компонентам. Иначе в распределенной транзакции будет использовано множество сессий, каждая из которых соответствует процессу или потоку на сервере базы данных, что может привести к резкому снижению производительности сервера СУБД.

Использование нескольких сессий в ADODB

Несмотря на то что модель объектов ADODB не предоставляет прямой возможности одновременного использования нескольких сессий с одним источником данных, это ограничение можно обойти. Решение основывается на применение внутреннего интерфейса ADOConnectionConstruction компоненты ADODB.Connection.

```
void clone_adodb_connection(IDispatch*    pCurrentConnection,
                        IDispatchPtr& spNewConnection)//throw
{
 //объявляем типы смарт-указателей для интерфейсов
 //конструирования ADODB-подключения
 DECLARE_IPTR_TYPE(ADOConnectionConstruction);
 DECLARE_IPTR_TYPE(ADOConnectionConstruction15);
 //1 получаем источник данных, привязанный к pCurrentConnection
 ADOConnectionConstructionPtr
    spConstruct(pCurrentConnection);
 ADOConnectionConstruction15Ptr
    spConstruct15(pCurrentConnection);
 if(!spConstruct && !spConstruct15)
  t_ole_error::throw_error("Объект – не ADODB.Connection",
                           E_INVALIDARG);
 IUnknownPtr spDataSource;
//берем указатель на OLE DB-источник данных
 if((bool)spConstruct15)
  spConstruct15->get_DSO(&spDataSource.ref_ptr());
```

```
 if(!spDataSource && (bool)spConstruct)
  spConstruct->get_DSO(&spDataSource.ref_ptr());
 if(!spDataSource)
  t_ole_error::throw_error(
         "ADODB.Connection не инициализирован",E_FAIL);
 //2 создаем новую сессию для spDataSource ------------------
 IUnknownPtr spNewSession;
 IDBCreateSessionPtr spDBCreateSession(spDataSource);
 assert((bool) spDBCreateSession);
 if(FAILED(spDBCreateSession->CreateSession
            (NULL,IID_IUnknown,&spNewSession.ref_ptr())))
 {
  t_ole_error::throw_error(
                 "Ошибка создания новой сессии",E_FAIL);
 }
 assert((bool)spNewSession);
 //3 создаем новый экземпляр ADODB.Connection ----------------
 IUnknownPtr spUnkNewConnection;
 HRESULT hr=SafeCreateInstance("ADODB.Connection",
                               NULL,CLSCTX_INPROC,IID_IUnknown,

(void**)&spUnkNewConnection.ref_ptr());
 if(FAILED(hr))
  t_ole_error::throw_error(
                 "Ошибка создания \"ADODB.Connection\"",hr);
 spConstruct=spUnkNewConnection;
 spConstruct15=spUnkNewConnection;
 if(!spConstruct15 && !spConstruct)
  throw t_ole_error(
           "Ошибка подключения к \"ADODB.Connection\"",E_FAIL);
 spDataSource->AddRef(); //ADO не вызывает для них AddRef
 spNewSession->AddRef();
 //при конструировании ADODB.Connection
 //пытается установить свои свойства,
 //в результате чего, если IBProvider уже подключен
 //к базе данных, может произойти ошибка
 //тем не менее подключение уже сконструировано
 //и вполне работоспособно.
 if((bool)spConstruct15) //IID__Connection15
 {
  spConstruct15->WrapDSOandSession(spDataSource,spNewSession);
  hr = spUnkNewConnection ->
            QueryInterface(IID__Connection15,spNewConnection);
 }
 else                    //IID__Connection
 {
  spConstruct->WrapDSOandSession(spDataSource,spNewSession);
  hr = spUnkNewConnection ->
            QueryInterface(IID__Connection,spNewConnection);
 }
 if(FAILED(hr))
  t_ole_error::throw_error(
```

```
    "Ошибка получения IDispatch из ADODB.Connection",hr);
 assert((bool)spNewConnection);
 //всё - spNewConnection подключен к тому же
 //источнику данных, но обладает
 //собственной сессией.
}//clone_adodb_connection
```

Чтение метаданных

Помимо управления транзакциями, сессия предоставляет еще одну полезную возможность – получение метаданных для базы данных (о метаданных см. главу "Структура базы данных InterBase" (ч. 4)). Поскольку в некоторых системах, например в Microsoft Distributed Query, операция получения метаданных выполняется очень часто, то IBProvider хранит информацию о них в оперативной памяти (т. е. кеширует). Кеширование метаданных можно настраивать для обеспечения оптимального быстродействия. Определить режим кеширования можно через свойство инициализации источника данных "schema_cache" и свойство сессии - "Session Schema Cache". Этим свойствам можно присваивать следующие значения:

- Кеширование запрещено. Данные будут всегда перечитываться.
- Глобальное кеширование на уровне Data Source. Это режим по умолчанию.
- Кеширование на уровне Session.

Если при запросе метаданных сессия содержит явно запущенную транзакцию, то провайдер не будет использовать дополнительную внутреннюю транзакцию для получения данных, а воспользуется уже существующей. Если явно запущенной транзакции нет, то IBProvider автоматически запустит внутреннюю транзакцию с уровнем изоляции, указанной в свойстве сессии "Autocommit Isolation Levels". Для запрещения автоматического запуска провайдером внутренних транзакций для извлечения метаданные необходимо определить в строке инициализации источника данных "inner_trans=false" или установить свойство сессии "Session InnerTrans=false".

Получение и вывод списка таблиц базы данных:

ADODB

```
Dim cn As New ADODB.Connection
Call cn.Open("file name=d:\database\employee.ibp")
Dim rs As ADODB.Recordset
Set rs = cn.OpenSchema(adSchemaTables)
Cells.Clear
Dim col As Long, row As Long
row = 1
'печать названия колонок
For col = 0 To rs.Fields.Count - 1
 Cells(row, col + 1) = rs(col).Name
Next col
'печать содержимого
While Not rs.EOF
 row = row + 1
```

```
 For col = 0 To rs.Fields.Count - 1
  Cells(row, col + 1) = rs(col).Value
 Next col
 rs.MoveNext
Wend
```

Здесь следует обратить внимание на одну особенность. Спецификация OLE DB для некоторых полей таблиц метаданных определяет типы, несовместимые с VARIANT, например UI8. Поэтому при попытке получения значения из этих полей через ADODB может возникнуть ошибка;

C++

```
try
{
 t_db_data_source cn;
 _THROW_OLEDB_FAILED(cn,attach("file
name=d:\\database\\employee.ibp"));
 t_db_session session;
 _THROW_OLEDB_FAILED(session,create(cn));
 //библиотека напрямую не поддерживает
 //работу с интерфейсом получения
 //наборов информационной схемы,
 //поэтому напишем необходимый код "в лоб".
 IDBSchemaRowsetPtr spSR(session.session_obj());
 if(!spSR)
  t_ole_error::throw_error(
        "query interface [IDBSchemaRowset]",spSR.m_hr);
 IUnknownPtr spUnk;
 HRESULT hr=spSR->GetRowset(NULL,DBSCHEMA_TABLES,0,NULL,
IID_IUnknown,0,NULL,&spUnk.ref_ptr());
 if(FAILED(hr))
  t_ole_error::throw_disp_error(hr,"get tables list");
 //подключаем полученный набор к курсору
 t_db_cursor cursor;
 _THROW_OLEDB_FAILED(cursor,attach(spUnk))
 //получаем описание полей результирующего
 //множества (набора данных)
 t_db_row row;
 _THROW_OLEDB_FAILED(cursor,describe(row))
 //печатаем содержимое набора
 while(cursor.fetch(row)==S_OK)
 {
  for(t_db_row::size_type i=0;i!=row.count;++i)
   cout<<row.columns(i).name<<":"<<row[i].as_string<<endl;
  cout<<endl;
 }//while
 //проверяем причину выхода из цикла
 _THROW_OLEDB_FAILED(cursor,m_last_result)
}
catch(const exception& exc){
 cout<<"error:"<<exc.what()<<endl;
}
```

Команда

Команда используется для выполнения SQL-запросов к базе данных. Важно не путать *команду*, которая является COM-объектом, с *текстом команды*, который представляет собой строку. Обычно команды используют для *описания данных*, например для создания таблицы и предоставления привилегий, и *манипуляции данными*, например для обновления и удаления строк. Особый случай *манипуляции данными* – создание набора строк (примером служит оператор SQL *SELECT*).

Спецификация OLE DB определяет гибкий набор интерфейсов для выполнения и обработки результатов SQL-запросов. Определяется так последовательность шагов:

1. Создание команды.
2. Установка текста запроса.
3. Подготовка запроса.
4. Подготовка параметров запроса.
5. Установка свойств результирующего множества (набора данных).
6. Выполнение запроса.

Создание команды

Как уже было отмечено ранее, объект команды создается объектом сессии. С помощью отдельной сессии можно создать много команд.

По умолчанию команда пользуется транзакцией породившей ее сессии. Если же сессия не содержит активной транзакции и разрешен режим автоматического подтверждения/отката, то команда будет выполняться в своей собственной автоматической транзакции, не зависящей от сессии.

Примеры создания команд:

ADODB

```
Dim cn As New ADODB.Connection
cn.Open "file name=d:\database\employee.ibp"
Dim cmd As New ADODB.Command
cmd.ActiveConnection = cn
```

C++

```
t_db_data_source cn;
 _THROW_OLEDB_FAILED(cn, attach(
    "file name=d:\\database\\employee.ibp"));
 t_db_session session;
 _THROW_OLEDB_FAILED(session,create(cn));
 t_db_command cmd1,cmd2;
 //Создаем команду традиционным способом,
 //используя объект C++ сессии
 _THROW_OLEDB_FAILED(cmd1,create(session));
 //Команду также можно создать, обладая
 //только указателем на IUnknown OLE DB-сессии.
```

```
//Это более подходящий способ для COM-объектов
IUnknownPtr spSession(session.session_obj());
_THROW_OLEDB_FAILED(cmd2,create(spSession));
```

Установка текста команды

Если команда только что создана, она еще не содержит текста. Поэтому текст SQL-запроса нужно установить.

Пример установки текста запроса:

ADODB

```
cmd.CommandText = "select * from job"
```

При этом IBProvider сбрасывает флаг подготовленности команды и очищает список параметров. Как это ни странно, здесь могут происходить ошибки:

1. Команда обнаружит смешанное использование параметров – именованных ":param" и неименованных, обозначаемых вопросительным знаком ("?").
2. Сбой преобразования SQL-запросов из ODBC-диалекта в вид, пригодный для передачи на сервер.

Все действия, связанные с установкой текста команды, осуществляются *локально*, т. е. обращение к серверу базы данных не производится.

Подготовка команды

Если пользователю *нужна информация о наборе рядов, который она создаст, то команду нужно подготовить*:

C++

```
t_db_row row;
_THROW_OLEDB_FAILED (cmd, prepare("select * from job",&row))
```

Поскольку с точки зрения взаимодействия с InterBase подготовка представляет собой передачу текста SQL-запроса серверу базы данных, то этот этап будет выполнен всегда – либо явным указанием пользователя, либо самой командой. При этом повторный вызов операции подготовки для одного и того же текста запроса игнорируется. Реализация команды провайдера для InterBase не осуществляет переподготовку запроса при повторном выполнении команды, поэтому явная подготовка с целью оптимизации многократного использования команды не имеет смысла.

ADODB способно самостоятельно определять необходимость явной подготовки команды, поэтому об этом можно не заботиться. Библиотека классов всегда проводит явную подготовку команды, выполняя ее сразу же после установки текста запроса.

Подготовка параметров SQL-запроса

Многократно выполняемые SQL-запросы, как правило, содержат *параметры*, представляющие собой переменные в тексте SQL-запроса. IBProvider поддерживает два вида параметров: именованные и неименованные. Перед выполнением

параметризованного SQL-запроса команда должна обладать описаниями параметров. *Описание параметра* – это его тип, имя, направление передачи значения (in-out). Пользователь может самостоятельно сформировать описания параметров или поручить формирование параметров команде.

Явное определение параметров SQL-запроса, несмотря на свою громоздкость при работе через ADODB, обеспечивает более эффективную работу с приложениями, поскольку исключается лишнее обращение к серверу для получения их описаний.

Пример явного определения параметров SQL-запроса:

ADODB

```
cmd.CommandText="select * from job where job_code=?"
cmd.Parameters.Append cmd.CreateParameter(,adBSTR,adParamInput)
cmd(0)="Eng"
```

C++

```
t_db_row row;
t_db_row param(1);
_THROW_OLEDB_FAILED(cmd2,
  prepare("select * from job where job_code=?",&row))
//тип параметра определяется его значением
param[0]="Eng";
param.count=1;
_THROW_OLEDB_FAILED(cmd2,execute(&param));
//тип параметра задается отдельно от значения,
//в этом случае провайдер выполнит преобразование значения
//в указанный тип.
set_param(param,0,adBSTR,"Eng");
param.count=1;
_THROW_OLEDB_FAILED(cmd2,execute(&param));
```

Автоматическое определение описаний параметров SQL-запроса позволяет клиентскому приложению перепоручить отслеживание типов параметров InterBase и конвертору типов IBProvider.

Пример явного указания команде сгенерировать описания типов:

ADODB

```
cmd.CommandText = "select * from job where job_code=?"
cmd.Parameters.Refresh
cmd(0) = "Eng"
```

Явное указание обновления списка параметров (cmd.Parameters.Refresh) обычно можно опустить. Однако иногда это необходимо. Например, для выполнения такого цикла:

ADODB

```
Dim cmd As New ADODB.Command
Dim rs As ADODB.Recordset
cmd.ActiveConnection = cn
cmd.CommandText = "select * from job where job_code=?"
Dim i As Long
For i = 0 To 10
```

```
 cmd.Parameters.Refresh
 cmd(0) = "Eng"
 Set rs = cmd.Execute
 '...
 'rs.Close
Next i
```

Вся хитрость заключается в том, что ADODB при выполнении второй итерации будет создавать новую OLE DB-команду, поскольку предыдущая занята обслуживанием результирующего множества SQL-запроса, созданного на первом шаге. Без строки cmd.Parameters.Refresh внутренний список описания параметров новой команды не будет сформирован, хотя коллекция ADODB.Command.Parameters будет содержать элементы. В результате при вызове метода cmd.execute в команду передаются значения параметров, описание которых у нее отсутствует. Принудительное обновление решает эту проблему. Понятно, что создание новой команды снижает производительность описанного выше алгоритма. Поэтому для того, чтобы ADODB могло повторно воспользоваться OLE DB-командой, нужно закрывать результирующее множество (rs.Close).

Повторный вызов cmd.Parameters.Refresh для одного и того же запроса не приводит к повторному обращению к серверу, поэтому расходы на такое дублирование ничтожны.

Автоматическая генерация описания параметров:

C++

```
_THROW_OLEDB_FAILED(cmd2,describe_params(param));
param[0]="Eng";
_THROW_OLEDB_FAILED(cmd2,execute(&param));
```

Существует единственное исключение, когда IBProvider обязательно выполнит дополнительный запрос на сервер для получения описания параметров SQL-запроса. Это касается случая, когда в параметре передается массив. Для такого типа параметров необходима дополнительная информация об имени таблицы и поля, в которые будут производить запись данных, а также информация о размерности массива. Подробности см. далее в разделе "Работа с массивами".

В вышеприведенных примерах были использованы неименованные параметры, обозначаемые в тексте запроса символом вопросительного знака. Именно такое обозначение параметров поддерживает и сам InterBase. Однако иногда удобно использовать именованные параметры в SQL-запросах:

- Именованный параметр можно многократно указывать в разных частях одного запроса.
- Порядок описания параметров может не соответствовать порядку использования параметров в тексте запроса. Это недопустимо для неименованных параметров.

В ADODB за удобство именованных параметров приходиться "платить" использованием режима автоматической генерации описания параметров (ADODB.Command.Parameters.Refresh). Причина заключается в том, что имя параметра, указываемое в ADODB.Command.CreateParameter, не передается команде. При использовании классов C++ такого ограничения нет – описание па-

раметров можно формировать обоими способами. Еще одним ограничением ADODB является невозможность использования именованных параметров для BLOB-полей – только неименованные параметры '?'.

Как уже было сказано выше, команда запрещает одновременное использование в тексте запроса именованных и неименованных параметров.

Сам InterBase поддерживает неименованные параметры. Поэтому команда вынуждена заменять в тексте запроса именованные параметры на неименованные параметры. Окончательный текст запроса, используемый для передачи на сервер, доступен через свойство команды "Prepare Stmt"

Пример многократного выполнения параметризованного запроса, содержащего именованный параметр:

ADODB

```
Dim cmd As New ADODB.Command
Dim rs As ADODB.Recordset
cmd.ActiveConnection = cn
cmd.CommandText = "select * from job where job_code=:job_code"
Dim i As Long
For i = 0 To 10
 cmd.Parameters.Refresh
 cmd("job_code") = "Eng"
 Debug.Print cmd.Properties("prepare stmt")
 Set rs = cmd.Execute
 '...
 rs.Close
Next i
```

Установка свойств результирующего множества

Перед выполнением команды, создающей набор рядов, можно задать различные свойства этого набора. Например, поддерживаемые интерфейсы, возможности позиционирования и верхняя граница объема оперативной памяти, используемой для хранения данных. Эти свойства задаются на уровне объекта команды и будут скопированы в набор свойств объекта результирующего множества.

Пример настройки свойств набора результирующего множества команды:

ADODB

```
'разрешить поддержку закладок для
'произвольного позиционирования в наборе
cmd.Properties("Use bookmarks") = True
'Использовать 1 MB памяти для кеширования рядов
cmd.Properties("Memory Usage") = 1024
```

C++

```
t_db_obj_props cmd_props(false);
_THROW_OLEDB_FAILED(cmd_props,
  attach_command(cmd2.command_obj()));
_THROW_OLEDB_FAILED(cmd_props,set("Use Bookmarks",true));
_THROW_OLEDB_FAILED(cmd_props,set("Memory Usage",1024));
```

Выполнение команды

Вызов операции execute является последним этапом выполнения SQL-запроса к базе данных, в котором участвует объект команды. Первоначальное и почти полное описание всех этапов выполнения этой операции заняло больше двух листов, забитых сухой технической информацией, дочитывая которую забываешь, с чего все началось. Шутка. Поэтому ограничимся коротким списком задач, выполняемых командой при вызове операции выполнения SQL-запроса.

- Проверка параметров. Количество разнообразных ошибок, вылавливаемых на этом этапе, превышает полтора десятка.
- Получение транзакции, в рамках которой будет выполняться SQL-запрос. Это может быть собственная активная транзакция родительской сессии или отдельная автоматически завершаемая транзакция (если таковые разрешены).
- Создание нового дескриптора низкоуровневого запроса, если текущий, принадлежащий команде, обслуживает набор строк, сформированных предыдущим вызовом операции execute. Такая ситуация может произойти при многократных вызовах execute для одного и того же SQL-запроса.
- Подготовка команды, если эта операция еще не была выполнена.
- Вызов InterBase API для выполнения запроса.
- Возвращение результата (если таковые создаются) через OUT-параметры или объект набора строк (rowset)

При выполнении SQL-запросов модификации данных (INSERT, UPDATE, DELETE) можно узнать число строк, затронутых запросом:

ADODB

```
cmd.CommandText =
"insert into project (proj_id,proj_name,proj_desc) " & _
                    "values(?,?,?)"
cmd(0) = 1001
cmd(1) = "test 1001"
cmd(2) = "test 1001"
Dim RowAffected As Long 'кол-во вставленных строк
cmd.Execute RowAffected
Debug.Print "insert " & CStr(RowAffected) & " rows"
```

Набор строк

Наборы строк – это центральные объекты, которые позволяют всем компонентам доступа к данным OLE DB представлять свои данные в табличной форме. Фактически набор строк – это совокупность строк, состоящих из полей данных. Компоненты доступа к базовым таблицам предоставляют свои данные в форме набор строк. Процессоры запросов (команда) представляют в форме набора строк результаты SQL-запросов. Это позволяет создавать слои объектов, поставляющих и потребляющих данные посредством одного и того же объекта.

СУБД InterBase поддерживает только однонаправленное движение курсора по набору строк, возвращаемому SQL-запросами. Под *курсором* здесь и далее

будет подразумеваться текущая позиция в наборе строк. Этого вполне достаточно для очень широкого круга задач. Положительной стороной однонаправленного обхода наборов строк в InterBase является возможность загрузки приложением большого объема данных без хранения в памяти уже обработанной информации. IBProvider по умолчанию обеспечивает именно такой способ "навигации" по множеству строк.

Хотя понятие курсора и присутствует в OLE DB, его основное назначение заключается в получении идентификаторов строк (HROW), а не самих данных полей строки. С помощью этих идентификаторов клиент может получать интересующие его данные в конкретной строке результирующего набора данных. То есть получив идентификатор строки, пользователь может выполнять многократное чтение полей одной и той же строки. Например, при первом чтении определяется размер данных BLOB-поля, а при втором осуществляется загрузка его содержимого.

Клиент обязан освободить идентификатор строки, когда последний ему уже не нужен. Но может это и не сделать, имитируя с помощью массива идентификаторов строк произвольный доступ к результирующему множеству SQL-запроса. При использовании ADODB этого можно добиться с помощью свойства ADODB.Recordset.CacheSize. В этом случае провайдер начнет осуществлять кеширование данных заблокированных рядов.

Помимо последовательного обхода всех строк множества, IBProvider поддерживает пропуск рядов и возможность возвращения курсора на начало множества за счет повторного выполнения запроса.

Пример создания набора строк, способа пропуска строк и возвращения курсора на начало набора:

ADODB

```
cmd.CommandText = "select * from job where job_code=:job_code"
cmd("job_code") = "Eng"
Set rs = cmd.Execute
'последовательный обход всех строк множества
While Not rs.EOF
  '…
  rs.MoveNext
Wend
rs.MoveFirst 'Restart
rs.Move 1 'пропускаем первый ряд
'...
rs.MoveFirst 'Restart
rs.Move 2 'пропускаем первые два ряда
'...
```

Произвольный доступ к результирующему множеству SQL-запросов IBProvider имитирует за счет кеширования выбранных данных на стороне клиента. Для работы в этом режиме провайдер использует более совершенный компонент управления множеством, реализующий возможности обратной выборки и произвольного перемещения по набору данных, а также возможность "приблизительного" позиционирования. И кроме того, в режиме произвольного доступа набор строк предоставляет *закладки строк,* с помощью которых клиент может

быстро возвращаться к некоторой строке. В некотором смысле закладки строк эквивалентны идентификатору строки (HROW), но гораздо более эффективны и не требуют никаких ресурсов для хранения. Кроме того, при работе через ADODB значение закладки текущей строки можно получить и сохранить для дальнейшего использования (см. ADODB.Recordset.Bookmark), а идентификатор строки – нет.

Ниже приведен пример создания набора строк, поддерживающего произвольный доступ, "программируя" его характеристики напрямую через свойства команды.

Пример позиционирования курсора набора рядов в случайном порядке:

ADODB

```
Dim cmd As New ADODB.Command
Dim rs As ADODB.Recordset
cmd.ActiveConnection = cn
cmd.CommandText = "select * from job where job_code=:job_code"
cmd("job_code") = "Eng"
'включаем поддержку закладок
cmd.Properties("Use Bookmarks") = True
Set rs = cmd.Execute
Dim i As Long
For i = 0 To rs.RecordCount
 'нумерация с единицы
 rs.AbsolutePosition = CLng(Rnd * rs.RecordCount) + 1
 '...
Next i
```

В общем, клиент независимо от режима доступа может заставить провайдер хранить данные выбранных строк. В первом случае это будет осуществлено за счет явного участия пользователя и провайдера, во втором случае – провайдер будет осуществлять это самостоятельно.

Поэтому набор строк OLE DB-провайдера для InterBase всегда использует собственный механизм контроля над объемом расходуемой памяти для хранения результирующего множества. Он заключается в удержании в указанном объеме памяти только наиболее часто используемых строк и вытеснении остальных строк во временный файл. При этом создание файла откладывается до последнего момента. Такой способ хранения данных выгодно отличает IBProvider от других компонентов доступа, которые основываются на поддержке со стороны ОС и использовании ее файла подкачки.

По умолчанию IBProvider удерживает в памяти 32 строки, независимо от их размера. Часто этого может оказаться недостаточно. Поэтому спецификация OLE DB определяет стандартное свойство набора строк "Memory Usage", которое позволяет отрегулировать верхнюю границу используемой памяти.

- 0 IBProvider удерживает в памяти 32 ряда. По умолчанию.
- 1...99 Использовать процент от доступной памяти ОС (как физической, так и файла подкачки).
- 100... Использование указанного размера памяти в килобайтах.

Естественно, нужно осознавать, что кеш результирующего набора данных в IBProvider является вторичным по отношению к файловому кешу ОС. Поэтому в случае экстремально больших размеров кеша в IBProvider вместо увеличения производительности можно получить прямо противоположный эффект.

Единственным ограничением, которое присутствует в текущей версии IBProvider (1.6.2), является невозможность редактирования выбранного множества строк, т. е. работы с так называемыми "живыми" запросами, которые часто используются в приложениях, создаваемых с помощью средств разработки компании Borland.

Практическое использование IBProvider

Работа с BLOB-полями

IBProvider предоставляет поддержку двум типам BLOB-полей: содержащих текст (SUB_TYPE TEXT) и бинарные данные. При этом доступ к BLOB может быть организован как к данным в памяти, так и к объекту-хранилищу. В любом случае провайдер не хранит сами данные BLOB-поля, а каждый раз загружает их по требованию клиента.

ADODB. Чтение BLOB:

```
Dim cn As New ADODB.Connection
cn.Open "file name=d:\database\employee.ibp"
cn.BeginTrans
Dim cmd As New ADODB.Command
Dim rs As ADODB.Recordset
cmd.ActiveConnection = cn
'JOB_REQUIREMENT - текстовое BLOB-поле
cmd.CommandText = "select job_requirement from job"
Set rs = cmd.Execute
'доступ к BLOB как к данным в памяти
While Not rs.EOF
 'печатаем размер BLOB-поля. обращение к ActualSize
 'не производит загрузки самих данных
 Debug.Print "size:" & CStr(rs(0).ActualSize)
 If IsNull(rs(0)) Then
  Debug.Print "NULL"
 Else
  Debug.Print rs(0)
 End If
 rs.MoveNext
Wend
Debug.Print "******************"
rs.MoveFirst
'чтение порциями по 40 байт
Dim seg As Variant, str As String
Const seg_size = 40
```

```
While Not rs.EOF
 'пропускаем пустые BLOB-поля
 If (Not IsNull(rs(0))) Then
  str = ""
  Do
   seg = rs(0).GetChunk(seg_size)
   'когда данных нет - GetChunk возвращает NULL
   If IsNull(seg) Then Exit Do
   str = str + seg
   Debug.Print "get chunk: " & Len(seg) & " bytes"
  Loop While True
  Debug.Print ">" & CStr(rs(0).ActualSize) & " - " & Len(str)
  Debug.Print ">" & str
 End If
 rs.MoveNext
Wend
cn.CommitTrans
```

ADODB. Запись BLOB-поля:

```
Dim cn As New ADODB.Connection
cn.Open "file name=d:\database\employee.ibp"
cn.BeginTrans
Dim cmd As New ADODB.Command
Dim rs As ADODB.Recordset
cmd.ActiveConnection = cn
'JOB_REQUIREMENT - текстовое BLOB-поле
cmd.CommandText = "select * from job"
Set rs = cmd.Execute
Dim upd_cmd As New ADODB.Command
upd_cmd.ActiveConnection = cn
upd_cmd.CommandText = _
  "update job set job_requirement=? " & _
  "where job_code=? and job_grade=? and " & _
        "job_country=?"
upd_cmd.Parameters.Refresh
Dim RowAffected As Long
While Not rs.EOF
 If (Not IsNull(rs("job_requirement"))) Then
  upd_cmd(0) = UCase(rs("job_requirement"))
  upd_cmd(1) = rs("job_code")
  upd_cmd(2) = rs("job_grade")
  upd_cmd(3) = rs("job_country")
  upd_cmd.Execute RowAffected
  Debug.Print "affect:" & CStr(RowAffected)
 End If
 rs.MoveNext
Wend
'отменяем все изменения в базе данных
cn.RollbackTrans
```

Работа с BLOB-полями через IBProvider на C++ также прозрачна. Хотя можно написать более интересные алгоритмы, например подстановка объекта-хранилища, полученного из результирующего множества, в качестве параметра в команду, привязанную к другой базе данных. Подробности см. в примерах из дистрибутива IBProvider и спецификации OLEDB – "BLOB's and OLE Objects".

Работа с массивами

Встроенная поддержка массивов является одним из основных пунктов списка достоинств InterBase как SQL сервера баз данных. И одновременно массивы возглавляют список его невостребованных возможностей. В практике сильная потребность в использовании массивов возникала только один раз. Эта работа была связана с анализом накапливаемой информации, касающейся функционирования торговой организации. Тогда основным препятствием оказалось отсутствие готового решения для работы с массивами InterBase из VBA (MS Office).

Поэтому в IBProvider была реализована поддержка массивов с предоставлением высокоуровневого представления этого несомненно полезного типа данных InterBase.

Особенности реализации поддержки массивов

- OLEDB-спецификация для представления типа данных "массив" использует структуру SAFEARRAY. Эта же структура употребления для управления массивами в Visual Basic.
- Элементы массивов не могут содержать NULL. Это ограничение связано с тем, что InterBase не поддерживает тип VARIANT.
- Все типы строк, хранящиеся в массивах, обрабатываются сервером как Си-строки, т. е. заканчивающиеся нулем. IBProvider исходит именно из такого способа хранения и не использует собственных символов типа \n для определения конца строки.
- Провайдер предоставляет полную поддержку для преобразования массивов из одного типа в другой. И пользуется ею по умолчанию, поскольку VB не понимает структуру SAFEARRAY, содержащую данные, несовместимые с VARIANT. Вообще говоря, наверное, можно было бы всегда возвращать массивы, содержащие VARIANT, но удалось обойтись без этой крайности. Отключить конвертирование массивов можно с помощью свойства инициализации источника данных (строка подключения) или набора строк "array_vt_type", установив его значение в false.
- Как и BLOB-поля, провайдер не хранит и не кеширует данные массива. Информация каждый раз загружается с сервера.
- Если при чтении массивов от пользователя не требуется никакой помощи, то для записи массивов провайдеру может потребоваться дополнительная информация. Дело в том, что запись массива, как и BLOB-поля, производиться отдельным обращением к InterBase API. Для этой операции требуется иметь описание, содержащее имя таблицы и имя колонки, в которую произ-

водится запись, тип элемента и сведения о размерности. Провайдер способен самостоятельно определить эту информацию только при работе с сервером InterBase 6.x и выше. Для работы с InterBase 4.x и InterBase 5.x IBProvider вводит нестандартное расширение, позволяющее определять в тексте запроса параметры вида "параметр.таблица.колонка". Этот синтаксис может быть использован как для именованных, так и для неименованных параметров:

```
update job set language_req=?.job.language_req
update job set language_req=:param.job.language_req
```

Переданные таким образом названия таблицы и колонки используются только для внутренних целей и недоступны вне провайдера. Если IBProvider смог самостоятельно получить эту информацию или значение параметра не является массивом, то пользовательская помощь игнорируется. В противном случае клиент отвечает за корректность переданных дополнительных сведений о параметре. Дополнительно заметим, что во 2-й и 3-й части такого составного имени параметра можно использовать квотированные названия объектов базы данных.

- Клиент не имеет возможности определить интересующее его подмножество массива. Провайдер всегда возвращает массив целиком. Это ограничение OLEDB, а не InterBase.
- При записи массива можно передавать подмножество. Но нужно учитывать, что массив всегда пересоздается. Поэтому в случае выполнения "UPDATE…", можно потерять предыдущую информацию из неуказанных элементов.

Пример чтения массивов.:

ADODB

```
Dim cn As New ADODB.Connection
cn.Open "file name=d:\database\employee.ibp"
Dim cmd As New ADODB.Command, rs As ADODB.Recordset
cmd.ActiveConnection = cn
cmd.CommandText = "select * from proj_dept_budget"
Set rs = cmd.Execute
Dim qhc As Variant ' QUART_HEAD_CNT
Dim i As Long
While Not rs.EOF
 If IsNull(rs("quart_head_cnt")) Then
  Debug.Print "NULL"
 Else
  qhc = rs("quart_head_cnt")
  For i = LBound(qhc, 1) To UBound(qhc, 1)
   Debug.Print "qhc[" & CStr(i) & "]=" & CStr(qhc(i))
  Next i
 End If
 rs.MoveNext
 Debug.Print "-----------------"
Wend
```

Пример записи массивов (InterBase 5.6):

ADODB

```
Dim cn As New ADODB.Connection
cn.Open "file name=d:\database\employee.ibp"
Debug.Print cn.Properties("IB Version")
cn.BeginTrans
Dim cmd As New ADODB.Command, rs As ADODB.Recordset
cmd.ActiveConnection = cn
cmd.CommandText = "select * from proj_dept_budget"
Set rs = cmd.Execute
Dim upd_cmd As New ADODB.Command
upd_cmd.ActiveConnection = cn
upd_cmd.CommandText = _
 "update proj_dept_budget " & _
 "set quart_head_cnt=:a.proj_dept_budget.quart_head_cnt " & _
 "where year=:year and proj_id=:proj_id and dept_no=:dept_no"
upd_cmd.Parameters.Refresh
Dim qhc As Variant ' QUAD_HEAD_CNT
Dim i As Long, RowAffected As Long
While Not rs.EOF
 If Not IsNull(rs("quart_head_cnt")) Then
  qhc = rs("quart_head_cnt")
  For i = LBound(qhc, 1) To UBound(qhc, 1)
   qhc(i) = 10 * qhc(i)
  Next i
  upd_cmd("a") = qhc
  upd_cmd("year") = rs("year")
  upd_cmd("proj_id") = rs("proj_id")
  upd_cmd("dept_no") = rs("dept_no")
  upd_cmd.Execute RowAffected ' транзакционные изменения
  Debug.Print ">" & CStr(RowAffected)
 End If
 rs.MoveNext
 Debug.Print "-----------------"
Wend
cn.CommitTrans
```

Работа с хранимыми процедурами

Хранимые процедуры делятся на две категории – селективные (процедуры-выборки) и исполняемые.

Принцип работы с селективной хранимой процедурой, возвращающей результат своей работы в виде набора строк, очень похож на выполнение обычного SQL-запроса "SELECT…" содержащего параметры.

Вызов селективной процедуры "SUB_TOT_BUDGET"

ADODB

```
'вспомогательная функция конвертирования VARIANT в строку
'с поддержкой NULL
 Function my_cstr(s As Variant) As String
```

```
 If (IsNull(s)) Then
  my_cstr = "NULL"
 Else
  my_cstr = CStr(s)
 End If
End Function
Sub sproc_select()
 Dim cn As New ADODB.Connection
 cn.Open "file name=d:\database\employee.ibp"
 cn.BeginTrans
 Dim cmd As New ADODB.Command
 cmd.ActiveConnection = cn
 cmd.CommandText = "select * from SUB_TOT_BUDGET(?)"
 cmd(0) = 100
 Dim rs As ADODB.Recordset
 Set rs = cmd.Execute
 Dim col As Long
 While Not rs.EOF
  Debug.Print "--------------"
   For col = 0 To rs.Fields.Count - 1
   Debug.Print CStr(col) & ":" & rs.Fields(col).Name & " - " &
my_cstr(rs(col))
  Next col
  rs.MoveNext
 Wend
 cn.CommitTrans
End Sub
```

Вызов исполняемой ХП отличается от использования селективной ХП в том плане, что применяют SQL-запрос EXECUTE PROCEDURE… и получают результат работы через out-параметры. IBProvider различает только вызов исполняемой ХП, анализируя сигнатуру SQL-запроса. Наряду с SQL-выражением EXECUTE PROCEDURE…, непосредственно поддерживаемого InterBase, в тексте команды можно указывать EXECUTE … и EXEC … IBProvider распознает в этих командах попытку вызова исполняемой ХП и автоматически приводит текст SQL-запроса к совместимому с InterBase. Для получения результата работы исполняемой ХП нужно либо самостоятельно описать out-параметры, либо попросить команду сформировать эти описания самостоятельно (ADODB.Command.Parameters.Refresh). Основными правилами здесь являются:

1. В тексте запроса out-параметры не упоминаются.
2. Описание out-параметров после in-параметров.
3. Не обязательно определять все out-параметры. При работе через ADODB, это могут быть первые out-параметры из всего списка (пропуски не допускаются). При прямой работе с OLE DB-командой можно указывать имена интересующих выходящих параметров ХП, тем самым получать out-параметры в любой комбинации.

Тестовая база данных employee.gdb не содержит готовых примеров исполняемых ХП, поэтому для следующего примера будет определена своя собственная простейшая хранимая процедура.

Определение исполнимой хранимой процедуры:

SQL

```
create procedure sp_calculate_values(x integer,y integer)
 returns(value1 integer,value2 varchar(64))
as
begin
 value1=x+y;
 value2=x-y;
end
```

Вызов и обработка результатов исполнимой хранимой процедуры SP_CALCULATE_VALUES:

ADODB

```
Sub sproc_exec()
 Dim cn As New ADODB.Connection
 cn.Open "file name=d:\database\employee.ibp"
 cn.BeginTrans
 Dim cmd As New ADODB.Command
 cmd.ActiveConnection = cn
 'автоматическое определение параметров
 cmd.CommandText = "exec sp_calculate_values(:x1,:x2)"
 cmd("x1") = 200
 cmd("x2") = 100
 cmd.Execute
 Debug.Print "out1=" & CStr(cmd("value1"))
 Debug.Print "out2=" & CStr(cmd("value2"))
 'явное определение параметров
 cmd.CommandText = "execute sp_calculate_values(?,?)"
 cmd.Parameters.Append cmd.CreateParameter(, adInteger,
    adParamInput, , 200)
 cmd.Parameters.Append cmd.CreateParameter(, adInteger,
    adParamInput, , 300)
 cmd.Parameters.Append cmd.CreateParameter("v1", adInteger,
    adParamOutput)
 cmd.Parameters.Append cmd.CreateParameter("v2", adBSTR,
    adParamOutput)
 cmd.Execute
 Debug.Print "v1=" & CStr(cmd("v1"))
 Debug.Print "v2=" & CStr(cmd("v2"))
 cn.CommitTrans
End Sub
```

При явном определении параметров можно попробовать испытать провайдер на "прочность", задав некорректный порядок перечисления параметров. Например, сначало out-параметры, потом in-параметры.

Создание COM-объектов для работы с базой данных

Самым эффективным применением технологии OLEDB является создание и использование специализированных компонентов для работы с базой данных. Этот подход изначально начал применяться для серверов приложений (application servers). Однако ничто не мешает реализовывать те же самые принципы при создании обычного приложения базы данных. В этом случае, помимо обычных преимуществ компонентной технологии, исчезают типичные проблемы, связанные с передачей и разделением ресурсов сервера баз данных между несколькими модулями клиентского приложения. Для малосвязанных модулей достаточно разделять одно подключение, в случае использования сервисных компонентов может потребоваться совместная работа в контексте одной транзакции. Поскольку эти ресурсы представлены в виде COM-объектов, то для корректной работы требуется только правильно управлять их счетчиком ссылок. Однако для опытного программиста, использующего COM-технологии в реальной работе, это не проблема.

Исходя из накопленного опыта создания таких компонентов и их использования из программ, написанных на C++, VBA и VBScript, можно порекомендовать следующую структуру COM-объектов:

- Дуальный (dual) интерфейс автоматизации, через который выполняется основное взаимодействие с объектом. Этот же интерфейс предоставляет свойство Connection для того, чтобы устанавливать и получать подключение, используя ADODB-компоненты. Как уже было сказано ранее, ADODB.Connection одновременно является и источником данных и сессией.
- Обычный интерфейс (наследующий IUnknown) для инициализации компонента посредством указателя на IUnknown сессии.
- Внутренняя работа с базой данных осуществляется через низкоуровневые интерфейсы OLEDB посредством классов C++. Таким образом, компонент изолируется от ADODB и обеспечивает более производительное функционирование собственных алгоритмов.

Принцип наиболее эффективной работы также не очень сложный – компонент должен свести к минимуму число создающихся и подготавливающихся команд. Так же имеет смысл загрузить в память содержимое таблиц, небольших по размеру и хранящих фиксированные данные. В основном под эту категорию попадают таблицы справочников. Тогда можно исключить из запросов все возможные обращения для выборки этой информации, что в конечном итоге, уменьшает нагрузку на сервер базы данных.

В качестве поддержки совместного использования ADODB и OLEDB в одном проекте инструментальная библиотека представляет две утилиты:

- construct_adodb_connection – создание ADODB подключения на базе существующего источника данных и сессии;
- get_adodb_session – получение OLEDB-сессии, обслуживаемой ADODB-подключением.

Несмотря на открывающиеся в связи с использованием IBProvider перспективы, связанные с дроблением ваших приложений для InterBase на модули, глав-

ное не переусердствовать. Не стоит делать компоненты, предназначенные для коллекций, с собственным механизмом чтения и записи. Помните, что любой использующий команды объект делает как минимум 4–5 обращений к серверу:

- Создание.
- Подготовка.
- Выполнение.
- Выборка результата.
- Разрушение.

Поэтому для групповых операций больше всего приемлем классический подход, когда реализуется групповая загрузка и запись, отделенная от самих данных.

Еще одной хорошей идеей является создание в приложении обычных классов с реализацией той логики, которая скорее всего не потребуется вне границ вашего приложения. Согласитесь, что написание класса и COM-компонента требует несравнимых усилий. Кроме того, не забывайте, что создание COM-объектов производиться через COM-инфраструктуру, поэтому накладные расходы распространяются и на время выполнения приложения. Поэтому обычные классы все равно остаются основным "тактическим средством" больших приложений, разработанных в объектно-ориентированном стиле.

Ниже приводиться пример COM-объекта, который подключается к сессии и используется для того, чтобы получить значение генератора (см. главу "Таблицы. Первичные ключи и генераторы" (ч. 1)). Здесь мы ограничимся лишь IDL-описанием двух интерфейсов и реализацией их методов. Помимо этого, используются возможности инструментальной библиотеки из дистрибутива IBProvider 1.6.2. В реальном случае этот код, конечно же, лучше оформить в виде обычного класса. Тогда можно исключить одновременную поддержку ADODB и OLEDB. Кроме того, в данном примере не оптимизирована работа метода GenID для случая повторного использования подготовленной команды, для случая многократного вызова метода с идентичными аргументами.

IDL-описание интерфейсов:

```
//////////////////////////////////////////////////////
//interface IDBSessionObject
// установка/получение рабочей OLEDB-сессии объекта
[
 object,
 uuid(98E5AB40-333E-11D6-AC8F-00A0C907DB93),
 pointer_default(unique)
]
interface IDBSessionObject:IUnknown
{
 HRESULT SetDBSession([in] IUnknown* pSession);
 HRESULT GetDBSession([out]IUnknown** ppSession);
};//interface IDBSessionObject
//////////////////////////////////////////////////////////
//interface IDBGenID
// интерфейс получения значения генератора
[
```

```
 object,
 uuid(98E5AB41-333E-11D6-AC8F-00A0C907DB93),
 dual,
 oleautomation,
 pointer_default(unique),
 nonextensible
]
interface IDBGenID:IDispatch
{
 [propput]
  HRESULT Connection([in]IDispatch* pConnection);
 [propget]
  HRESULT Connection([out,retval]IDispatch** ppConnection);
  HRESULT Convert([in]BSTR GenName,
                  [in]LONG Count,
                  [out,retval]LONG*   pResult);
};//interface IDBGenID
```

Реализация методов установки сессии:

```
//m_spADODBConnection  - член класса,
//  содержащий указатель на ADODB-подключение
//m_spSession - член класса,
//  содержащий указатель на используемую OLEDB-сессию
//m_Cmd - команда (t_db_command) получения значения генератора
//IDBSessionObject interface ------------------------------
HRESULT __stdcall TDBGenID::SetDBSession(IUnknown* pSession)
{
 ::SetErrorInfo(0,NULL);
 HRESULT hr=S_OK;
 _OLE_TRY_
 {
  //освобождаем ADODB connection
  m_spADODBConnection.Release();
  m_spSession=pSession;
  //инициализируем объекты взаимодействия с базой данных
  m_Cmd.destroy();
 }
 _OLE_DISP_CATCHES_
 return hr;
}//SetDBSession
HRESULT __stdcall TDBGenID::GetDBSession(IUnknown** ppSession)
{
 ::SetErrorInfo(0,NULL);
 return m_spSession.CopyTo(ppSession);
}//GetDBSession
//IOC2_ObjectLoader interface -----------------------------
HRESULT __stdcall TDBGenID::put_Connection
                              (IDispatch* pConnection)
{
 ::SetErrorInfo(0,NULL);
 HRESULT hr=NOERROR;
 _OLE_TRY_
```

```
 {
  IDispatchPtr spConnection(pConnection);  //блокируем в памяти
  //освобождаем текущие подключения
  SetDBSession(NULL);
  if(pConnection)
  {
   IUnknownPtr  spDBSession;
   get_adodb_session(pConnection,spDBSession); //throw
   if(SUCCEEDED(hr=SetDBSession(spDBSession)))
    m_spADODBConnection=pConnection;
  }//pConnection!=NULL
 }
 _OLE_DISP_CATCHES_
 return hr;
}//put_Connection
HRESULT __stdcall TDBGenID::get_Connection
                              (IDispatch** ppConnection)
{
 ::SetErrorInfo(0,NULL);
 if(ppConnection==NULL)
  return E_POINTER;
 *ppConnection=NULL;
 HRESULT hr=S_OK;
 _OLE_TRY_
 {
  if(!m_spADODBConnection && (bool)m_spSession)
  {
   IGetDataSourcePtr spGetDataSource(m_spSession);
   if(!spGetDataSource)
    t_ole_error::throw_error
     ("query IGetDataSource interface",spGetDataSource.m_hr);
   IUnknownPtr spDataSource;
   if(FAILED(hr=get_data_source(spGetDataSource,spDataSource)))
    t_ole_error::throw_error("Получение источника данных",hr);
   IDBPropertiesPtr spDBProperties(spDataSource);
   if(!spDBProperties)
    t_ole_error::throw_error
      ("query IDBProperties interface",spDBProperties.m_hr);
   construct_adodb_connection(spDBProperties,m_spSession,
                              m_spADODBConnection);//throw
  }//if - создание ADODB-объекта
  hr=m_spADODBConnection.CopyTo(ppConnection);
 }
 _OLE_DISP_CATCHES_
 return hr;
}//get_Connection
```

Реализация метода получения значения генератора:

```
HRESULT __stdcall TDBGenID::GenID(BSTR GenName,LONG Count,
                                  LONG* pResult)
{
 ::SetErrorInfo(0,NULL);
```

```
 if(pResult==NULL)
  return E_POINTER;
 HRESULT hr=S_OK;
 _OLE_TRY_
 {
  if(!m_spSession)
   throw runtime_error("Объект неинициализирован");
  if(!m_Cmd.is_created())
   _THROW_OLEDB_FAILED(m_Cmd,create(m_spSession));
  structure::str_formatter stmt
   ("select gen_id(%1,%2) from rdb$database");
  t_db_row row(1);
  _THROW_OLEDB_FAILED(m_Cmd,prepare(stmt<<GenName<<Count,&row))
  _THROW_OLEDB_FAILED(m_Cmd,execute(NULL));
  if(m_Cmd.fetch(row)==S_OK)
   *pResult=row[0].as_integer;
  else
  {
   //проверим причину сбоя получения данных
   _THROW_OLEDB_FAILED(m_Cmd,m_last_result)
   throw runtime_error("Получено пустое множество");
  }
 }
 _OLE_DISP_CATCHES_
 return hr;
}//GenID
```

Использование скриптов в клиентских приложениях базы данных InterBase

Время от времени у любого программиста появляется желание вынести часть логики своих приложений на уровень, который можно было бы изменять без перекомпиляции приложения. А для определенного класса задач это требование изначально является первоочередным. Как правило, когда речь заходит о добавлении такой возможности, сразу вспоминают серверы приложений. Однако существуют задачи, для которых то же самое эффективнее реализовывать на уровне отдельного клиента.

Использование такого рода "программ" разгружает основной код приложения от алгоритмов, сильно подверженных переменчивым желаниям пользователей. Например:

- Проверка достоверности данных при сохранении.
- Контроль прав на чтение и изменение данных.
- Правила движения документов.
- Настраиваемый пользовательский интерфейс.
- Печать выходных форм.
- Встроенные отчеты.

При этом не существует больших проблем с выполнением сценариев, поскольку реализовано и доступно достаточно много компонентов для быстрого

решения этой задачи. В том числе и бесплатный ActiveX-компонент ScriptControl от Microsoft.

Основные трудности при реализации такого подхода приходятся на осуществление тесной интеграции основного кода приложения и кода сценария. В числе важных вопросов находится и обеспечение доступа сценария к базе данных.

Самым тривиальным решением было бы создание внутри сценария собственного подключения к база данных. Но, как уже было замечено ранее, это медленно и неэффективно. Другим решением является передача готового подключения из основного кода приложения. И вот здесь применение ADODB- и OLEDB-компонентов доступа дает максимальный эффект.

Компоненты ADODB изначально приспособлены для использования в ActiveX-сценариях. Впрочем, для передачи в сценарий соединения с базой данных ADODB.Connection с выделенной сессией могут потребоваться некоторые усилия. Решение этой задачи было приведено в разделе описания сессии.

Компоненты OLEDB нельзя непосредственно использовать в сценариях. Но их можно "обернуть" в ADODB-компоненты и таким образом использовать в коде сценария. Подробности можно посмотреть в примерах, входящих в дистрибутив IBProvider.

Естественно, что для сложных задач проблема связи сценариев с базами данных не является основной. Тем не менее стоимость программного решения может резко возрасти, если приложение будет базироваться на компонентах доступа, которые нельзя ни напрямую, ни через какой-либо адаптер использовать в сценариях.

Использование пула подключений к базе данных

Для многопользовательских серверных приложений, обрабатывающих клиентские запросы с помощью запросов к отдельной базе данных, повторное употребление ресурсов SQL-сервера является одним из основных способов увеличения производительности. Поэтому пул подключений, кеширующий инициализированные источники данных, является важной составляющей такого рода программного обеспечения. Кроме того, важно понимать, что механизм пула подключений не реализуется самим OLE DB-провайдером. От последнего требуется только корректно обрабатывать уведомления о помещении в пул и обеспечивать многопоточный доступ к компонентам. IBProvider реализует оба требования, поэтому клиенту предлагается только провести корректную инициализацию источника данных.

Для включения пула подключений при работе через ADODB нужно указать в строке подключения параметры

```
"OLE DB Services=-1;free_threading=true"
```

Параметр "OLE DB Services=-1" указывает ADODB на необходимость использования пула подключений. Параметр "free_threading=true" устанавливает внутренний флаг, объявляющий поддержку многопоточного доступа.

Для демонстрации работы пула ниже приведен простой пример, выполняющий в цикле инициализацию и закрытие источника данных. Для запрещения пу-

ла подключений присвойте "OLE DB Services" нулевое значение. Замеры производительности проводятся очень грубо – в секундах, но этого оказалось достаточно, чтобы увидеть преимущества пула подключений. По истечении 60 с с момента добавления в пул неиспользуемые источники данных освобождаются и производиться отключение от сервера базы данных

Для того чтобы подробно изучить процесс функционирования пула подключений, следует воспользоваться информацией на сайте компании Microsoft или посмотреть в документацию по OLE DB SDK (см. "Resource Pooling").

ADODB:

```
Dim cn As New ADODB.Connection
Dim cnt As Long
Dim start As Date, total As Date
total = Time
For cnt = 1 To 10
 start = Time
 cn.Provider = "LCPI.IBProvider.1"
 cn.Properties("OLE DB Services") = -1
 cn.Properties("free_threading") = True
 cn.Open "data source=localhost:d:\database\employee.gdb;", _
         "gamer", "vermut"
 Dim cmd As New ADODB.Command
 cmd.ActiveConnection = cn
 cmd.CommandText = "select count(*) from job"
 cn.BeginTrans
 cmd.Execute
 cn.CommitTrans
 cn.Close
 Debug.Print ">" & CStr(CDate(Time - start))
 'можно сделать задержку чуть больше 60 с,
 'чтобы понаблюдать за освобождением
 'инициализированного источника данных  и
 'потерю подключения к базе данных
 'Application.Wait Time + CDate("0:1:05")
 'Debug.Print "disconnect"
 'Application.Wait Time + CDate("0:0:15")
Next cnt
Debug.Print "total:" & CStr(CDate(Time - total))
'освобождение последнего объекта, использующего
'пул подключений, приводит к уничтожению всех
' инициализированных источников данных
Set cn = Nothing
```

Распределенные запросы

Помимо определения самой спецификации OLE DB, Microsoft активно применяет ее в своих разработках, связанных с управлением данными. И одной из самых потрясающих разработок этой категории является Microsoft Distributed Query – программный компонент, входящий в состав MS SQL, позволяющий

делать SQL-запросы к нескольким источникам данных с использованием OLE DB-провайдеров. И хотя возможность обращения в одном запросе сразу к нескольким источникам данных так же доступна и в BDE, процессор распределенных SQL-запросов, реализованный Microsoft, несомненно, представляет собой более мощный и более совершенный механизм для этих целей. Далее будут перечислены основные моменты и принципы использования IBProvider в распределенных запросах с применением MS SQL 7.

1. MS Distributed Query потребовал полной стабильности в описании метаданных. Для этого пришлось реализовать в IBProvider полную поддержку всех типов InterBase и обеспечить совпадение описания метаданных в наборах информационной схемы с описанием колонок результирующих множеств.
2. Из-за скрупулезной сверки данных результирующих множеств с описанием их метаданных не допускается усечение хвостовых пробелов полей типа CHAR. По умолчанию усечение производится. Чтобы запретить эту операцию, в строке подключения к базе данных нужно указать свойство инициализации источника данных "truncate_char=false".
3. Процессор распределенных запросов не поддерживает массивы, поэтому не стоит их выбирать в результирующее множество.
4. Несовпадение диапазона дат MS SQL и InterBase приводит к тому, что нельзя выбирать даты до 1 января 1753 года.
5. При работе с 3-м диалектом подключения нужно соблюдать регистр символов имени объекта базы данных, независимо от того, квотировано оно или нет. Дело в том, что процессор запросов начинает повсеместно использовать двойные кавычки для имен объектов базы данных независимо от того, хотите вы этого или нет. Для подключения 1-го диалекта квотированные имена не используются, поэтому большие и маленькие символы в названии объектов базы данных не различаются.

Регистрация базы данных к процессору распределенных запросов MS SQL 7:

1. Скопируйте и зарегистрируйте IBProvider на компьютер с сервером баз данных MS SQL 7, процессор запросов которого будет использоваться для выполнения распределенных запросов.
2. Откройте "SQL Server Enterprise Manager" (консоль управления MS SQL серверами) и подключитесь к интересующему вас серверу.
3. Перейдите на "Security\Linked Servers" дерева элементов конфигурирования сервера. В контекстном меню этого элемента выберите пункт "New Linked Server ..."
4. В открывшемся диалоге нужно указать параметры подключения к базе данных InterBase через IBProvider. Для этого:
   - В поле "Linked Server" укажите имя, которое будет использоваться в SQL-запросах для идентификации нашей базы данных. Например, "IB_EMP".
   - В выпадающем списке "Provider Name" выберите "LCPI OLE DB Provider for InterBase".

- Нажмите кнопку Options и установите галочку напротив пункта "Allow InProcess". Закройте это окно, нажав кнопку OK.
- В поле "Data Source" нужно указать путь к базе данных. Например, main:e:\database\employee.gdb.
- В поле "Provider string" указываются остальные параметры подключения: "`use id=gamer; password=vermut; free_threading=true; truncate_char=false`".
- Закройте диалог ввода параметров подключения.

Если все параметры были введены правильно, то, перейдя в дереве на элемент "Security\Linked Servers\IB_EMP\Tables", можно посмотреть на список таблиц базы данных "employers.gdb". Если при указании параметров подключения была допущена ошибка, связанный сервер нужно удалить (выбрав в его контекстном меню "Удалить") и повторить операцию регистрации с самого начала.

После того как источник данных был подключен в качестве связанного сервера к MS SQL, можно начать эксперименты с SQL-запросами. Для этого нужно открыть SQL Server Query Analyzer. Это можно сделать из меню "Tools" консоли управления MS SQL-серверами или через меню группы программ для работы с MS SQL, которое доступно через кнопку "Пуск" панели задач Windows.

В поле ввода SQL-запросов Query Analyzer можно вводить как одиночный текст запроса, так и группу запросов, используя точку с запятой в качестве разделителя. При наборе запросов нужно использовать синтаксис MS SQL, а не InterBase, поскольку последний переходит в ранг обычного носителя данных взаимодействие с которым осуществляется через OLE DB-интерфейсы и очень простые SQL-запросы.

Как уже было описано ранее, любая операция с данными базы InterBase, требует наличия активной транзакции. Поэтому самым первым SQL-запросом будет команда запуска транзакции:

MS SQL

```
BEGIN TRANSACTION;
```

После этого можно вводить SQL-запросы на выборку данных из источника данных IB_EMP:

MS SQL

```
SELECT EMP.* FROM IB_EMP...EMPLOYEE EMP WHERE
EMP.JOB_CODE='Eng';
```

где IB_EMP…EMPLOYEE представляет собой составное имя объекта, которое полностью идентифицирует его расположение в пространстве имен сервера баз данных MS SQL.

Теперь можно зарегистрировать любую другую базу данных InterBase, выполнив действия по аналогии с вышеописанными. В том числе никто не запрещает зарегистрировать несколько раз одну и ту же базу данных под разными псевдонимами, например IB_EMP и IB_EMP_1. Конечно, в качестве связанного сервера может использоваться любой другой OLE DB-поставщик данных, который поддерживает необходимую для процессора запросов функциональность.

Пример нахождения записей описания служащих источника данных IB_EMP_1, отсутствующих в базе данных IB_EMP:

MS SQL

```
SELECT EMP1.*
FROM IB_EMP_1...EMPLOYEE EMP1
WHERE NOT EXISTS(SELECT * FROM IB_EMP...EMPLOYEE EMP
                 WHERE EMP1.FIRST_NAME=EMP.FIRST_NAME AND
                       EMP1.LAST_NAME=EMP.LAST_NAME)
```

Если IB_EMP и IB_EMP_1 являются псевдонимами одной и той же базы, то результирующее множество этого SQL-запроса должно быть пустым.

При первом обращении к связанному источнику данных его сессия подключается к координатору распределенной транзакции, активизированному командой "BEGIN TRANSACTION" и для дальнейшей работы будет использоваться транзакция, принадлежащая сессии. В предыдущем примере для выполнения запроса к координатору транзакций будет подключено две сессии, каждая из которых содержит активную транзакцию. Поэтому, теоретически, если между запусками этих двух транзакций таблица EMPLOYEER будет изменена, результирующее множество может оказаться непустым. На практике, конечно, с одной базой через два различных псевдонима обычно не работают.

Поскольку в настоящий момент IBProvider (версия 1.6.2) не реализует OLE DB-интерфейсы модификации результирующих множеств, использование процессора запросов для выполнения запросов "INSERT …", "UPDATE …", "DELETE…" невозможно.

Заключение

В этой главе рассмотрены способы разработки клиентских приложений для InterBase 6.x и его клонов с помощью технологии OLEDB. Показаны основные подходы к разработке приложений баз данных InterBase с использованием таких популярных средств разработки компании Microsoft, как Visual Basic и Visual C++. Мощные и гибкие возможности, предоставляемые IBProvider, опровергают распространенное мнение, что InterBase – это сервер, с которым можно работать только с помощью средств разработки от компании Borland.

Создание CGI-приложений под ОС Linux с использованием InterBase API

(Материал для данной главы предоставлен Сергеем Ивановичем Мереуцей и публикуется с его любезного разрешения.)

Подавляющее количество CGI-приложений ориентировано на чтение данных и отображение полученных результатов в виде HTML. В общем виде вся работа приложения может быть описана следующей схемой:

1. Чтение переменных, специфичных для WWW (разбор переменных html-форм и т. д.).
2. Подключение к базе данных.
3. Некоторое количество выборок и вставок (возможно, с исполнением хранимых процедур).
4. Отключение от базы данных.
5. Отправка данных клиенту.

Следует отметить, что часто пп. 4 и 5 меняются местами, например, при отображении данных в виде HTML-таблиц, построении отчетов и в прочих ситуациях, когда данных, полученных из базы данных, может быть много и хранение их в оперативной памяти вплоть до разрыва соединения нецелесообразно.

Также необходимо помнить, что, как правило, "тяжелые" запросы, которые могут выполняться несколько минут, в CGI-приложениях не используются; в случае употребления таких запросов браузер клиента считает, что произошел сбой соединения и сообщает об истечении времени ожидания.

Еще одной особенностью CGI-приложений является то, что практически всегда известно количество, наименование и тип входных и выходных параметров. Также обычно не возникает необходимости в обработке исключительных ситуаций, таких, как разрыв соединения во время работы с базой данных, так как пользователь, увидев в своем браузере пустой или некорректный результат, обычно ее обновляет, что дает возможность вновь выполнить запрос.

CGI-приложение является по сути прослойкой между Интернетом и базой данных, поэтому особое внимание следует уделять корректной работе с памятью и данными, которые приложение будет получать из Интернента. Самый лучший вариант – считать, что изначально будут посылаться некорректные данные, поэтому придется разработать эффективную защиту от "замусоривания".

При работе на уровне API дополнительный разбор текстовых переменных не производится – они попадают в базу данных в том виде, в каком хранятся в памяти скрипта. Таким образом, можно, например, хранить текст в различных кодировках в полях таблиц с CHARACTER SET NONE. Также не возникает проблем с одинарными или двойными кавычками.

Весь процесс создания CGI-приложения можно условно разделить на две части: сначала в общем виде создается шаблон HTML-документа, который должен получиться в результате работы, а затем необходимо реализовать собственно CGI-скрипт.

На начальном этапе написания приложения достаточно заменить переменные, которые в рабочей версии будут выбираться из присылаемых браузером запросов, на константы. Обычно всю отладочную информацию также выводят в результирующий HTML-документ – это позволяет легко отлаживать приложение. После того как основной алгоритм программы заработает, можно вместо констант обрабатывать непосредственно переменные из HTTP-запросов.

Специфика CGI-приложений предполагает, что в качестве SQL-сервера хорошо проявит себя архитектура SuperServer, которая обеспечивает хорошую производительность на множестве маленьких запросов при относительно небольшом объеме занимаемой памяти.

CGI-приложения можно писать на любом языке программирования – от C до Perl, где есть соответствующие библиотеки для работы с WWW.

Давайте рассмотрим написание CGI-приложения под ОС Linux с использованием языка C/C++, который производит откомпилированный код и избавляет от необходимости распространять приложения в исходных кодах, как в случае Perl или PHP. Все примеры в данной главе написаны на C/C++, однако легко могут быть переписаны под FPC/PERL/PHP.

В комплект поставки InterBase и его клонов входит заголовочный файл ibase.h с описанием функций InterBase API, констант, макросов и т. д., доступных разработчику клиентских приложений InterBase на C/C++. Написание приложений с использованием чистого API может показаться непривычно громоздким из-за высокой детализации кода приложения. Зная работу с базой данных на уровне InterBase API, можно довольно легко написать свою обертку вокруг API-функций или воспользоваться уже существующими (например, IBProvider из предыдущей главы).

Прежде чем перейти к рассмотрению реальных примеров, необходимо ввести следующие понятия:

Параметры запроса – это переменные, которые передаются серверу (входящие) или возвращаются в качестве результата (исходящие). В примерах эти переменные носят соответственно названия isqlda и osqlda.

Большинство IB API-функций возвращают так называемый *STATUS_VECTOR* – массив целых чисел (long). Анализируя этот массив, можно узнать, произошел ли вызов функции успешно, и если нет – то по какой причине.

В качестве параметров используется особая структура, XSQLDA – eXtended SQL Descriptor Area. Она позволяет узнать (или передать) всю необходимую информацию о каждом конкретном параметре, а также сами данные в структуре SQLVAR – SQL Variable. В силу изложенного выше при написании CGI-приложений интерес представляют лишь несколько полей из этих структур:

Структура XSQLDA, описывающая параметры запросов

Поле	Тип (для языка C)	Описание
version	short	Версия XSQLDA-структуры; должна быть установлена в соответствии с версией используемой клиентской библиотеки. Для текущих версий InterBase это значение соответствует SQDA_VERSION1

Поле	Тип (для языка C)	Описание
sqln	short	Количество элементов массива sqlvar (иными словами, количество передаваемых и/или получаемых переменных). Это поле должно быть обязательно заполнено для входящих и исходящих параметров до передачи данных запроса серверу
sqld	short	Указывает количество входящих или исходящих параметров. Обычно это поле для исходящих параметров инициализируется библиотекой InterBase. 0 в этом поле означает, что запрос не является выборкой (non-SELECT query)

Структура SQLVAR, описывающая данные параметров структуры XSQLDA

Поле	Тип (для языка C)	Описание
sqldata	char *	Указатель на данные, которые необходимо передать серверу
sqltype	short	Информация о типе передаваемых данных. Для полей, допускающих SQL-значение NULL-используется тип, на единицу больший, чем базовый. Например, для типа SQL INTEGER используется значение SQL_LONG, если NULL не допускается, и тип SQL_LONG+1, если в значении поля допускается NULL
sqllen	short	Размер памяти, занимаемой переменной, на которую указывает sqldata
sqlind	short *	Это поле – индикатор того, является ли значение передаваемой переменной NULL (в этом случае это поле должно быть равно единице) или нет (в этом случае значение этого поля 0). Заполнение этого поля является обязательным, если переменная может принимать значение NULL. Таким образом, если вы хотите передать значение NULL серверу и в sqltype указан не базовый тип (например, SQL_LONG+1), то инициализировать это поле обязательно; если же указан, например тип SQL_LONG, то это поле можно не инициализировать – сервер проигнорирует его значение. При чтении данных из результата это поле указывает, является ли полученное значение NULL или отличается от него. Анализировать необходимо именно это поле – результат сравнения переменной sqldata с пустым значением может

Поле	Тип (для языка C)	Описание
		быть абсолютно непредсказуемым

Подробнее все поля этих структур описаны в [7] и представляют интерес только при написании действительно сложных приложений или библиотек.

Теперь, когда даны необходимые определения, можно непосредственно познакомиться с методами DSQL-программирования. Их всего 4:

Нет переменных ни на входе ни на выходе. В этом случае обе структуры SQLDA (isqlda, osqlda) должны быть инициализированы NULL (или любым пустым указателем в используемом вами языке – например, nil в Паскале) в вызове функций isc_dsql_execute, isc_dsql_execute2 и т. д. Обычно это запросы, которые удаляют записи с жестко заданными свойствами.

- Есть переменные как на входе, так и на выходе; в этом случае нужно инициализировать соответствующие структуры.
- Нет переменных на выходе, но есть на входе; в этом случае osqlda = NULL, а isqlda требуется инициализировать.
- Нет переменных на входе, но есть на выходе; в этом случае isqlda = NULL, а osqlda требуется инициализировать.

Теперь примеры. Для простоты будем работать только с одной таблицей, структура которой описана ниже.

Пример 1. Запрос без параметров

Предположим, у нас есть таблица следующей структуры:

```
CREATE TABLE BOOKS (
    B_ID INTEGER NOT NULL,
    B_INDEX CHAR(16) NOT NULL,
    B_NAME VARCHAR(80) NOT NULL,
    B_AUTHOR VARCHAR(80) NOT NULL,
    B_ADDED TIMESTAMP DEFAULT 'now' NOT NULL,
    B_THEME VARCHAR(60) NOT NULL);
```

Для того чтобы отобразить результат запроса "select b_id, b_index, b_name, b_author, b_added, b_theme from books" в виде следующей таблицы (см. Рис. 3.1):

Example Nr 1
SELECT without input parameters

Book ID	CODE	TITLE	AUTHOR	ADDED	THEME
1	AAAAAAAAAAAAAAAA	Как приготовить суп	Иванов	21-Mar-2002 00:00	Кулинария

Рис. 3.1. Результат выполнения SQL-запроса, представленный в браузере в виде HTML-таблицы

понадобится следующий скрипт: (example1.c)

```
#include <ibase.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <time.h>
// Эта структура предназначена для хранения переменных типа SQL_VARYING
#define SQL_VARCHAR(len) struct {short vary_length; char
vary_string[(len)+1];}
int main (void){
// Константы, необходимые для работы с базой данных – инициализируйте их в
// соответствии с реальным путем к базе, пользователем и паролем
char *dbname = "localhost:/var/db/demo.gdb";
char *uname = "sysdba";
char *upass = "masterkey";
char *query = "select b_id, b_index, b_name, b_author, b_added,
b_theme from books";
// Переменные для работы с базой данных
isc_db_handle                    db_handle = NULL;
isc_tr_handle                    transaction_handle = NULL;
isc_stmt_handle                  statement_handle=NULL;
char                          dpb_buffer[256], *dpb, *p;
short                         dpb_length;
ISC_STATUS                    status_vector[20];
XSQLDA                        *isqlda, *osqlda;
long                          fetch_code;
short
      o_ind[20]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
// Остальные переменные
int                           i=0;
long                          b_id;
char                          b_index[17];
SQL_VARCHAR(100)              b_name;
SQL_VARCHAR(100)              b_author;
SQL_VARCHAR(100)              b_theme;
ISC_TIMESTAMP                 b_added;
struct tm                     added_time;
char                          decodedTime[100];
```

Здесь наше приложение начинает вывод стандартного HTML-документа:

```
printf("Content-type:
text/plain\n\n<html><body><center><b>Example Nr 1</b><br>SELECT
without input parameters</center><br>");
```

а затем подключается к базе данных – здесь два этапа:

```
// создаем так называемый database parameter buffer, необходимый
// для подключения к базе данных
dpb=dpb_buffer;
*dpb++ = isc_dpb_version1;
*dpb++ = isc_dpb_user_name;
*dpb++ = strlen(uname);
for(p = uname; *p;) *dpb++ = *p++;
*dpb++ = isc_dpb_password;
*dpb++ = strlen(upass);
for (p=upass; *p;) *dpb++ = *p++;
dpb_length = dpb- dpb_buffer;
```

```
// Подключаемся к базе
isc_attach_database(
status_vector,
strlen(dbname),
dbname,
&db_handle,
dpb_length,
dpb_buffer);
```

Далее идет стандартная для большинства API-функций проверка и анализ результата:

```
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
}
```

Если подключение к базе данных произошло удачно, начинается транзакция:

```
if (db_handle){
isc_start_transaction(
status_vector,
&transaction_handle,
1,
&db_handle,
0,
NULL);
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
}
}
```

Далее инициализируются структуры, которые будут заполняться результатами запроса:

```
osqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(6));
 osqlda -> version = SQLDA_VERSION1;
 osqlda -> sqln = 6;
 osqlda->sqlvar[0].sqldata = (char *)&b_id;
 osqlda->sqlvar[0].sqltype = SQL_LONG;
 osqlda->sqlvar[0].sqlind  = &o_ind[0];
 osqlda->sqlvar[1].sqldata = (char *)&b_index;
 osqlda->sqlvar[1].sqltype = SQL_TEXT;
 osqlda->sqlvar[1].sqlind  = &o_ind[1];
 osqlda->sqlvar[2].sqldata = (char *)&b_name;
 osqlda->sqlvar[2].sqltype = SQL_VARYING;
 osqlda->sqlvar[2].sqlind  = &o_ind[2];
 osqlda->sqlvar[3].sqldata = (char *)&b_author;
 osqlda->sqlvar[3].sqltype = SQL_VARYING;
 osqlda->sqlvar[3].sqlind  = &o_ind[3];
 osqlda->sqlvar[4].sqldata = (char *)&b_added;
 osqlda->sqlvar[4].sqltype = SQL_TIMESTAMP;
 osqlda->sqlvar[4].sqlind  = &o_ind[4];
 osqlda->sqlvar[5].sqldata = (char *)&b_theme;
```

```
 osqlda->sqlvar[5].sqltype = SQL_VARYING;
 osqlda->sqlvar[5].sqlind  = &o_ind[5];
```

А вот здесь, собственно, и начинается подготовка к исполнению запроса сервером:

```
isc_dsql_allocate_statement(
status_vector,
&db_handle,
&statement_handle);
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
 }
isc_dsql_prepare(
status_vector,
&transaction_handle,
&statement_handle,
0,
query,
SQL_DIALECT_V6,
osqlda);
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
 }
isc_dsql_execute2(
status_vector,
&transaction_handle,
&statement_handle,
1,
NULL,
NULL);
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
 }
```

Здесь начинается таблица HTML-документа. Ситуация, когда в базе данных может не оказаться данных, подробно анализируется во втором примере.

```
printf("<center><table border=0 bgcolor=black cellpadding=1
cellspacing=1><tr align=center bgcolor=#999999> <td>Book
ID</tr> <td>CODE</tr> <td>TITLE</tr> <td>AUTHOR</tr>
<td>ADDED</tr> <td>THEME</tr> </tr>");
```

После исполнения запроса сервер готов к передаче данных. "Доставкой" данных занимается функция isc_dsql_fetch():

```
while((fetch_code = isc_dsql_fetch(
status_vector,
&statement_handle,
1,
osqlda))==0){
```

Для строковых переменных требуется корректно установить длину, так как размер возвращаемых данных не всегда соответствуем максимально возможному, и если этого не сделать, то вместе с реальными данными можно получить "мусор" из памяти или остатки предыдущих строк:

```
b_index[osqlda->sqlvar[1].sqllen]='\0';
b_name.vary_string[b_name.vary_length]='\0';
b_author.vary_string[b_author.vary_length]='\0';
b_theme.vary_string[b_theme.vary_length]='\0';
```

Структуру типа TIMESTAMP, как и структуры DATE/TIME, перед выводом в документ можно преобразовать в строковый тип в нужном формате. Для этого сначала она декодируется в структуру tm, а затем в строку:

```
isc_decode_timestamp(&b_added,&added_time);
strftime(decodedTime,sizeof(decodedTime),"%d-%b-%Y
%H:%M",&added_time);
printf("<tr bgcolor=white><td>%i</td> <td>%s</td> <td>%s</td>
<td>%s</td> <td>%s</td> <td>%s</td> </tr>",
b_id,
b_index,
b_name.vary_string,
b_author.vary_string,
decodedTime,
b_theme.vary_string);
}
```

После вывода всех данных необходимо завершить документ:

```
printf("</table></center></body></html>");
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
 }
free(osqlda);
isc_dsql_free_statement(
status_vector,
&statement_handle,
DSQL_drop);
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
 }
```

Затем завершить транзакцию и отключиться от базы данных:

```
if (transaction_handle){isc_commit_transaction(status_vector,
&transaction_handle);}
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
 }
if (db_handle) isc_detach_database(status_vector, &db_handle);
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
```

```
return(1);
}
return(0);
}// end of main
```

Обратите внимание на маленькую разницу в работе с переменными SQL_VARYING и SQL_TEXT (это соответственно VARCHAR и CHAR языка SQL). Разница в том, что если в базе данных хранится меньше символов, чем максимально возможно для столбца (например, объявлено CHAR(16), а хранится строка "12345"), то сервер добавит N пробелов в конец строки, где N является разницей между максимально возможным количеством символов и реально хранящимся в поле таблицы. Тип SQL_VARYING свободен от этого недостатка, однако при получении данных нужно учитывать, что только определенное значение символов является реально полученными; остальное количество – это случайные данные из памяти компьютера, на котором исполняется скрипт. Для удобства работы с этим типом обычно определяют структуру, где член структуры SQL_VARCHAR vary_length указывает размер полученной строки, а vary_string собственно содержит строку.

Если запрос гарантированно возвращает одно значение (например, одиночный SELECT или вызов хранимой процедуры), то использовать функцию isc_dsql_fetch() нет необходимости; вместо этого в параметр функции isc_dsql_execute2() можно подставить значение osqlda переменной. Работа с типами SQL DATE и TIME абсолютно не отличается от работы с переменными типа TIMESTAMP – всего лишь используются другие функции для преобразования: isc_decode_sql_date() и isc_decode_sql_time().

Пример 2. Запрос с параметрами

Теперь рассмотрим пример исполнения запроса с параметрами – вызов хранимой процедуры, которая просто вставит данные из формы в эту же таблицу. Принципиально этот пример практически ничем не отличается от вышеприведенного, за исключением того что в нем появляются две дополнительные части – одна разбирает переменные HTML-формы, другая (если переменные переданы) исполняет процедуру.

Вот текст этой ХП:

```
create procedure InsertData (b_index char(16),
b_name varchar(80),
b_author varchar(80),
b_theme varchar(60))
returns (result_code integer)
as
begin
insert into books (
 B_ID,B_INDEX,B_NAME, B_AUTHOR,B_ADDED,B_THEME)
 values(0,:b_index, :b_name, :b_author, 'now', :b_theme);
 result_code = 0;
when any
do begin
result_code=-1;
```

```
end
end
```

Текст ХП достаточно банальный, вместо него в действительности можно было бы воспользоваться командой INSERT, однако подразумевается, что в реальной процедуре производятся некоторые манипуляции с входными данными (например, код книги может генерироваться не генератором, а по определенному алгоритму) и в качестве результата либо происходит вставка данных, либо процедура возвращает код ошибки.

Текст скрипта второго примера выглядит так :

```
#include <ibase.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "cgic.h"
#define SQL_VARCHAR(len) struct {short vary_length; char
vary_string[(len)+1];}
```

Вот здесь некоторое отличие: используемая для разбора переменных www-библиотека заменяет стандартную функцию main:

```
int cgiMain (void){
char *dbname = "localhost:/var/db/demo.gdb";
char *uname = "sysdba";
char *upass = "masterkey";
char *query = "select b_id, b_index, b_name, b_author, b_added,
b_theme from books";
```

На месте неизвестных входящих параметров – знаки вопроса:

```
char *SPCall = "execute procedure insertdata(?,?,?,?)";
isc_db_handle                    db_handle = NULL;
isc_tr_handle                    transaction_handle = NULL;
isc_stmt_handle                  statement_handle=NULL;
char                          dpb_buffer[256], *dpb, *p;
short                         dpb_length;
ISC_STATUS                    status_vector[20];
XSQLDA                        *isqlda, *osqlda;
Long                          fetch_code;
Short
      o_ind[20]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
int                           i=0;
int                           hDisplayed=0;
int                           formVarErr;
int                           ExecSP=1;
long                          res_code;
long                          b_id;
char                          b_index[17];
SQL_VARCHAR(100)              b_name;
SQL_VARCHAR(100)              b_author;
SQL_VARCHAR(100)              b_theme;
ISC_TIMESTAMP                 b_added;
struct tm                     added_time;
```

```
char                            decodedTime[100];
char                            form_b_index[17],
form_b_name[81],
form_b_author[81],
form_b_theme[61];
```

В этой части происходит анализ переменных, полученных скриптом, и в зависимости от метода вызова принимается решение, исполнять ли ХП или нет.

```
printf("Content-type:
text/plain\n\n<html><body><center><b>Example Nr 2</b><hr
width=80\%></center><br>   Part 1: Executable SP
demo with both types of parameters.<br>");
if(strcmp(cgiRequestMethod,"POST")==0){
formVarErr = cgi-
FormStringNoNewlines("b_index",form_b_index,17);
if (formVarErr!=cgiFormSuccess){
printf("<br><b>Error: Book index missed or too long</b>");
ExecSP=0;
}
formVarErr = cgiFormStringNoNewlines("b_name",form_b_name,81);
if (formVarErr!=cgiFormSuccess){
printf("<br><b>Error: Book name missed or too long</b>");
ExecSP=0;
}
formVarErr = cgi-
FormStringNoNewlines("b_author",form_b_author,81);
if (formVarErr!=cgiFormSuccess){
printf("<br><b>Error: Book author missed or too long</b>");
ExecSP=0;
}
formVarErr = cgi-
FormStringNoNewlines("b_theme",form_b_theme,61);
if (formVarErr!=cgiFormSuccess){
printf("<br><b>Error: Book theme missed or too long</b>");
ExecSP=0;
}
}
else{
ExecSP=0;
printf("<br><i>Procedure execution skipped</i> - REQUEST_METHOD
must be POST");
}
```

Заметьте: если специфика приложения требует того, чтобы определенные www-переменные обязательно присутствовали и удовлетворяли определенным условиям (например, учетные данные пользователя могут быть защищены контрольной суммой), то именно в этом месте приложение производит решение о целесообразности продолжения работы.

После разбора переменных идет непосредственно работа с базой данных:

```
dpb=dpb_buffer;
*dpb++ = isc_dpb_version1;
```

```
*dpb++ = isc_dpb_user_name;
*dpb++ = strlen(uname);
for(p = uname; *p;)
  *dpb++ = *p++;
*dpb++ = isc_dpb_password;
*dpb++ = strlen(upass);
for (p=upass; *p;)
  *dpb++ = *p++;
dpb_length = dpb- dpb_buffer;
isc_attach_database(
status_vector,
strlen(dbname),
dbname,
&db_handle,
dpb_length,
dpb_buffer);
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
}
if (db_handle){
isc_start_transaction(
status_vector,
&transaction_handle,
1,
&db_handle,
0,
NULL);
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
}
}
```

Если были получены данные и они корректны, происходит вызов хранимой процедуры:

```
if(ExecSP){
printf("<br><i>Attempt to call SP with the following parame-
ters:
'%s','%s','%s','%s'</i>.....",form_b_index,form_b_name,form_b_a
uthor,form_b_theme);
```

Как можно видеть, принципиально инициализация структур для входящих параметров не сильно отличается от инициализации исходящих параметров из первого примера:

```
isqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(4));
 isqlda->version = SQLDA_VERSION1;
 isqlda->sqln = 4;
 isqlda->sqld = 4;
 isqlda->sqlvar[0].sqldata = (char *)&form_b_index;
 isqlda->sqlvar[0].sqltype = SQL_TEXT;
 isqlda->sqlvar[0].sqllen  = strlen(form_b_index);
```

```
 isqlda->sqlvar[1].sqldata = (char *)&form_b_name;
 isqlda->sqlvar[1].sqltype = SQL_TEXT;
 isqlda->sqlvar[1].sqllen  = strlen(form_b_name);
 isqlda->sqlvar[2].sqldata = (char *)&form_b_author;
 isqlda->sqlvar[2].sqltype = SQL_TEXT;
 isqlda->sqlvar[2].sqllen  = strlen(form_b_author);
 isqlda->sqlvar[3].sqldata = (char *)&form_b_theme;
 isqlda->sqlvar[3].sqltype = SQL_TEXT;
 isqlda->sqlvar[3].sqllen  = strlen(form_b_theme);
osqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(1));
 osqlda -> version = SQLDA_VERSION1;
 osqlda -> sqln = 1;
 osqlda -> sqld = 1;
 osqlda->sqlvar[0].sqldata = (char *)&res_code;
 osqlda->sqlvar[0].sqltype = SQL_LONG;
osqlda->sqlvar[0].sqllen  = sizeof(long);
 osqlda->sqlvar[0].sqlind  = &o_ind[0];
```

Вызов ХП происходит сразу, без предварительной подготовки (подробнее об этом будет рассказано несколько ниже):

```
isc_dsql_exec_immed2(
status_vector,
&db_handle,
&transaction_handle,
0,
SPCall,
SQL_DIALECT_V6,
isqlda,
osqlda);
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
}
if(res_code==0){
printf(" <b>successfly</b>.");
}
else{
printf(" <b>failed with result_code=%i</b>.",res_code);
}
 free(isqlda);
 free(osqlda);
}
```

Далее идет выборка данных из таблицы базы данных – это несколько модифицированная часть из первого примера, которая анализирует, были ли получены данные или таблица оказалось пустой и отображать нечего:

```
printf("<br><br>   Part 2: Select without input
parameters<br>");
osqlda = (XSQLDA *)malloc(XSQLDA_LENGTH(6));
 osqlda -> version = SQLDA_VERSION1;
 osqlda -> sqln = 6;
 osqlda->sqlvar[0].sqldata = (char *)&b_id;
 osqlda->sqlvar[0].sqltype = SQL_LONG;
```

```
 osqlda->sqlvar[0].sqlind  = &o_ind[0];
 osqlda->sqlvar[1].sqldata = (char *)&b_index;
 osqlda->sqlvar[1].sqltype = SQL_TEXT;
 osqlda->sqlvar[1].sqlind  = &o_ind[1];
 osqlda->sqlvar[2].sqldata = (char *)&b_name;
 osqlda->sqlvar[2].sqltype = SQL_VARYING;
 osqlda->sqlvar[2].sqlind  = &o_ind[2];
 osqlda->sqlvar[3].sqldata = (char *)&b_author;
 osqlda->sqlvar[3].sqltype = SQL_VARYING;
 osqlda->sqlvar[3].sqlind  = &o_ind[3];
 osqlda->sqlvar[4].sqldata = (char *)&b_added;
 osqlda->sqlvar[4].sqltype = SQL_TIMESTAMP;
 osqlda->sqlvar[4].sqlind  = &o_ind[4];
 osqlda->sqlvar[5].sqldata = (char *)&b_theme;
 osqlda->sqlvar[5].sqltype = SQL_VARYING;
 osqlda->sqlvar[5].sqlind  = &o_ind[5];
isc_dsql_allocate_statement(
status_vector,
&db_handle,
&statement_handle);
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
}
isc_dsql_prepare(
status_vector,
&transaction_handle,
&statement_handle,
0,
query,
SQL_DIALECT_V6,
osqlda);
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
}
```

Заметьте, что если бы запрос гарантированно возвращал одну запись, то надобность в вызове функции isc_dsql_fetch()отпала бы, а на месте последних двух пустых значений в вызове нижеследующей функции были бы соответственно isqlda и osqlda:

```
isc_dsql_execute2(
status_vector,
&transaction_handle,
&statement_handle,
1,
NULL,
NULL);
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
}
while((fetch_code = isc_dsql_fetch(
status_vector,
```

```
&statement_handle,
1,
osqlda))==0)
 {
```

Вот здесь и производится проверка на существование хотя бы одной записи в таблице:

```
if (!hDisplayed){
hDisplayed=1;
printf("<br><br><center><table border=0 bgcolor=black cellpad-
ding=1 cellspacing=1><tr align=center bgcolor=#999999> <td>Book
ID</tr> <td>CODE</tr> <td>TITLE</tr> <td>AUTHOR</tr>
<td>ADDED</tr> <td>THEME</tr> </tr>");
}
```

Далее идет уже знакомая по предыдущему примеру обработка текстовых переменных:

```
b_index[osqlda->sqlvar[1].sqllen]='\0';
b_name.vary_string[b_name.vary_length]='\0';
b_author.vary_string[b_author.vary_length]='\0';
b_theme.vary_string[b_theme.vary_length]='\0';
```

и преобразование TIMESTAMP в удобочитаемое выражение:

```
isc_decode_timestamp(&b_added,&added_time);
strftime(decodedTime,sizeof(decodedTime),"%d-%b-%Y
%H:%M",&added_time);
printf("<tr bgcolor=white><td>%i</td> <td>%s</td> <td>%s</td>
<td>%s</td> <td>%s</td> <td>%s</td> </tr>",
b_id,
b_index,
b_name.vary_string,
b_author.vary_string,
decodedTime,
b_theme.vary_string);
}
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
}
free(osqlda);
isc_dsql_free_statement(
status_vector,
&statement_handle,
DSQL_drop);
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
}
```

Завершаем транзакцию и отключаемся от базы данных:

```
if (transaction_handle){
isc_commit_transaction(status_vector, &transaction_handle); }
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
}
```

```
if (db_handle) isc_detach_database(status_vector, &db_handle);
if (status_vector[0] == 1 && status_vector[1]){
isc_print_status(status_vector);
return(1);
}
```

Завершаем документ HTML-формой для ввода данных:

```
if(hDisplayed){
 printf("</table></center><br>");
}
else{
printf("<br><br><br><center><b>Table is emp-
ty</b></center><br><br><br>");
}
printf("<center><table border=0 width=45%><tr
align=right><td><form name='demo' method=POST ac-
tion='example2'><table border=0>");
printf("<tr><td align=left>Book index</td><td><input type=text
name='b_index' size=16 maxlength=16></td></tr>");
printf("<tr><td align=left>Book name</td><td><input type=text
name='b_name' size=16 maxlength=80></td></tr>");
printf("<tr><td align=left>Book author</td><td><input type=text
name='b_author' size=16 maxlength=80></td></tr>");
printf("<tr><td align=left>Book theme</td><td><input type=text
name='b_theme' size=16 maxlength=60></td></tr>");
printf("<tr><td colspan=2 align=right><input type='submit' val-
ue='Add data'></td></tr>");
printf("</table></form></td></tr></table></center></body></html
>");
return(0);
}// end of main
```

Вот что мы получим при первом запуске приложения при пустой таблице данных (см. рис 3.2):

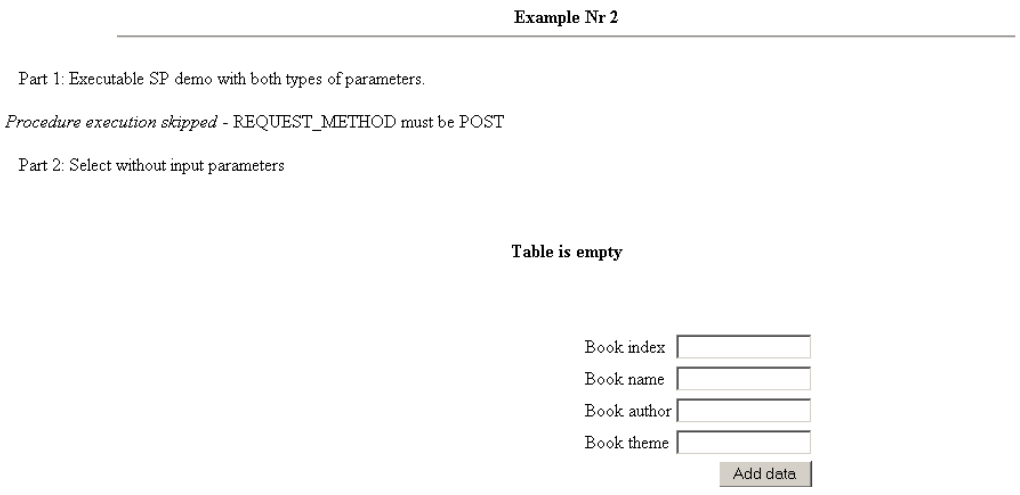


Рис. 3.2. Результат вставки записи

После вставки какой-нибудь записи в таблицу при помощи формы справа снизу результат будет следующим (см. рис 3.3).

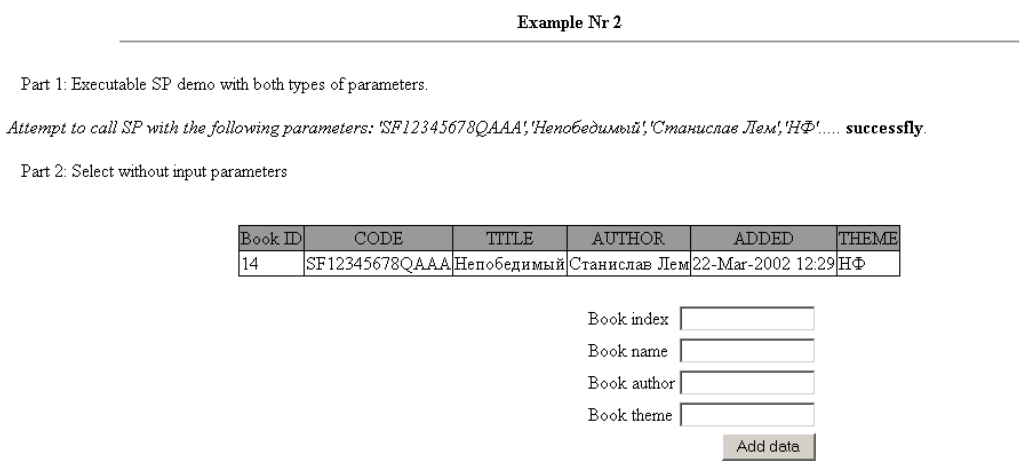


Рис. 3.3. Результаты выполнения CGI-приложения

В этом приложении читается несколько переменных www-окружения, которые проверяются на ненулевую длину и передаются в качестве входящего параметра запроса. Переменные www-окружения читаются при помощи функций библиотеки CGIC (ее можно загрузить с сайта http://www.boutell.com/cgic), однако вы можете воспользоваться любой удобной вам библиотекой. В качестве результата анализируется переменная res_code, и в зависимости от ее значения сообщается результат вызова процедуры. Неизвестные входные параметры, участвующие в запросе, передаются следующим образом: на местах неизвестных параметров ставится знак вопроса (?), а в соответствующей этому параметру переменной XSQLDA определяется тип и данные для передаваемой переменной, причем допускается смешивать родственные типы (int-smallint, char-varchar и т. д.). При внимательном изучении примера видно, что запрос, выбирающий данные из таблицы, вызывается несколько отличным методом, нежели запрос вызывающий ХП. Первый запрос, состоит из следующих этапов:

1. Резервирование ресурсов, требуемых для запроса библиотекой доступа к InterBase, – вызов функции isc_dsql_allocate_statement().
2. Подготовка запроса к исполнению сервером – именно на этом этапе выдаются сообщения об ошибках в синтаксисе – вызов функции isc_dsql_prepare().
3. Исполнение запроса сервером – вызов функции isc_dsql_execute2() (или isc_dsql_execute(), отличие этих функций в том, что вызов isc_dsql_execute() непременно требует вызова функции isc_dsql_fetch(), если существуют исходящие параметры). На этом этапе могут быть выданы сообщения о несоответствии типов и/или количества элементов XSQLDA.
4. Доставка данных приложению – вызов isc_dsql_fetch(). В отличие от большинства остальных API-функций эта функция возвращает целое число, содержащее

код ошибки, – пока это значение равно нулю, данные есть и их можно прочитать. Когда выбраны все данные, это значение становится отличным от нуля.

5. Освобождение ресурсов, занятых на этапе 1, – вызов isc_dsql_free_statement().

Инициализация структур XSQLDA обычно происходит до п. 2.

Второй запрос исполняется сразу, минуя все вышеперечисленные пункты, кроме п. 3. В данном случае используется функция isc_dsql_exec_immed2(). Отличие данных методов заключается в том, что в случае первого запроса можно воспользоваться пп. 3 и 4 внутри цикла – это дает преимущество в скорости, при исполнении одного и того же запроса, но с разными значениями параметров, так как синтаксис уже проверен и запрос сразу исполняется. Метод, реализованный в исполнении второго запроса, обычно применяется при однократном вызове хранимых процедур или при исполнении команд, которые разрешается исполнять только в этих функциях (например, CREATE DATABASE разрешается использовать только в вызове функции isc_dsql_execute_immediate()).

Так как в случае "немедленного" исполнения запроса не происходит его предварительного анализа, то необходимо инициализировать переменную sqld структуры XSQLDA, которая используется для получения результата (в данном случае это делает строка osqlda -> sqld = 1). Если не проинициализировать эту переменную, то в качестве результата вызова функции isc_dsql_exec_immed2 будет получена ошибка Message length error… По той же причине, необходимо явно указывать размер памяти, который будет занимать переменная, хранящая результат, – в данном случае это sizeof(long). Если этого не сделать, то последствия будут непредсказуемы, – в лучшем случае вы не получите ничего на выходе.

При разработке CGI-приложений, работающих с InterBase, следует придерживаться следующих правил:

- Все переменные www-окружения должны быть проанализированы до подключения к базе данных – может случиться, что нужных данных нет или они не удовлетворяют каким-либо требованиям и работа с базой данных заведомо теряет смысл. В этом случае следует проинформировать пользователя о неверно введенных данных.
- Старайтесь вывод данных, не зависящих от работы с самой базой данных, производить вне подключения, тем самым сохраняя ресурсы сервера.
- Запускайте, если это возможно, сервер от имени пользователя с ограниченными правами.
- Создайте клиента базы данных (см. главу "Безопасность в InterBase: пользователи, роли и права" (ч. 4)), которого будут использовать все www-приложения для доступа к базе данных, и назначьте ему минимально необходимые права.

Маленький совет напоследок (применимый для InterBase с архитектурой SuperServer): если прописать вызов сервера в inittab с параметром respawn, то система сама перезапустит сервер в случае его падения надежнее, чем это сделает guardian. Таким образом, получится некий аналог птицы Феникс – сервер базы данных возродится сразу же после фатальной ошибки (допущенной, например, при разработке приложений на API).

Заключение

Несмотря на устрашающий размер примеров, в работе с базами данных InterBase и его клонов на уровне API нет ничего слишком сложного, скорее там много монотонной работы по кодированию. Однако приложения с использованием InterBase API являются при правильном написании наиболее производительными.

Поэтому изучайте примеры в этой книге- и создавайте on-line-игры, www-магазины и порталы!

Работа с InterBase с использованием ODBC

(Материал данной главы предоставлен Алексеем Сергеевичем Карякиным и публикуется с его любезного разрешения.)

Интерфейсы ODBC основаны на международном стандарте ISO/IEC 9075-3:1995 Information technology -- Database languages -- SQL -- Part 3: Call-Level Interface (SQL/CLI). Стандартизация интерфейсов доступа к данным позволяет разрабатывать "горизонтальные" приложения, не зависящие на уровне исходного кода от используемой базы данных.

Термин "ODBC" является сокращением английских слов "Open DataBase Connectivity" и обозначает набор интерфейсов прикладного уровня (API – Application Programming Interface), предоставляющих возможность обращения к базам данных из приложений. Кроме интерфейсов, ODBC фирмы Microsoft предоставляет инфраструктуру компонентов доступа к данным.

В этой главе описано использования драйвера Gemini InterBase ODBC для подключения приложений к базам данных InterBase с точки зрения пользователя, а также особенности данного драйвера. Если вы программируете с использованием интерфейса ODBC, вам следует ознакомиться документацию The ODBC Programmer's Reference, входящую в состав Microsoft Developer Network Library (MSDN).

Gemini InterBase ODBC driver соответствует версии 3.51 спецификации Microsoft ODBC, поддерживаются все функции уровней Level 0 (Core), Level 1 и большинство функций Level 2. Высокая степень соответствия стандарту позволяет использовать большинство приложений, поддерживающих интерфейс ODBC, среди которых:

- пакет Microsoft Office;
- генератор отчетов Seagate Crystal Reports;
- приложения, написанные с использованием технологии ADO, в том числе приложения ASP для Microsoft IIS;
- Microsoft SQL Server (использование баз данных InterBase как связанного сервера).

Драйвер существует в двух вариантах – настольном (desktop) и серверном (site). Первый вариант предназначен для офисных приложений, генераторов отчетов и других приложений, используемых на клиенте. Второй вариант рассчитан на использование в составе серверов приложений или Web-серверов.

Возможности драйвера Gemini ODBC

Поддержка кодировки UNICODE

Microsoft ODBC 3.5 определяет два типа драйверов – ANSI и UNICODE. Gemini ODBC-драйвер является по этой классификации драйвером UNICODE. Это дает возможность приложениям, использующим версию UNICODE интерфейса ODBC, обрабатывать данные различных национальных наборов символов.

Для хранения таких данных InterBase предоставляет кодировку (character set) UNICODE_FSS, но вы также можете использовать другие кодировки при хранении данных, в любом случае текстовые строки будут переданы в приложение правильно.

Вызов хранимых процедур InterBase с использованием стандартного синтаксиса ODBC

Как известно, InterBase использует два типа хранимых процедур: так называемые selectable-процедуры и executeable-процедуры; при этом процедуры разного типа отличаются способом вызова в SQL. В отличие от других ODBC-драйверов, Gemini ODBC отслеживает тип процедуры и всегда формирует корректный SQL-вызов без дополнительных пользовательских настроек.

Прокручиваемые курсоры

Gemini ODBC-драйвер поддерживает наравне с однонаправленными (FORWARD-ONLY) курсорами также статические (STATIC) необновляемые курсоры.

Асинхронная отмена вызовов для InterBase 6.5

Начиная с версии 6.5 Gemini ODBC-драйвер способен использовать новую возможность InterBase версии 6.5 – асинхронную отмену выполняющихся на сервере запросов.

Настройка используемого диалекта InterBase SQL

Gemini ODBC драйвер поддерживает настройку диалекта SQL клиентского приложения. В зависимости от диалекта драйвер определяет возможности сервера и сообщает их приложению через соответствующие функции ODBC API. Например, в диалекте 3 InterBase поддерживает quoted identifiers, и поэтому, при работе с базой данных через приложение SQL Explorer, тексты SQL запросов будут формироваться с идентификаторами в кавычках.

Gemini ODBC-драйвер поддерживает все типы данных в каждом из диалектов, включая NUMERIC/DECIMAL, DATE, TIME и TIMESTAMP.

Настройка параметров транзакций

Опции настройки DSN предусматривают задание параметров транзакций: использование команд COMMIT/ROLLBACK или COMMIT RETAINING/ROLLBACK RETAINING при завершении транзакции, установку режима "только чтение", установка режима ожидания (WAIT/NO_WAIT) и запрещение выборки старых версий при уровне изоляции READ COMMITTED.

Установка драйвера и настройка источников данных

Дистрибутив драйвера состоит из одного исполнимого файла с именем ibgem_21_desk.exe (для настольной редакции драйвера версии 2.1). Чтобы установить драйвер, необходимо запустить этот файл.

Существует два способа создания соединений в ODBC – с использованием DSN (Data Source Name – имя источника данных) и без DSN (так называемые DSN-less-соединения.

В первом случае все параметры соединения (такие, как имя базы данных, сервер и сетевой протокол) конфигурируются пользователем и хранятся в отдельном ключе системного реестра для каждого DSN. При соединении приложение указывает имя DSN, а также, возможно, имя пользователя и пароль для аутентификации. Источники данных бывают системные (System DSN), пользовательские (User DSN), а также файловые (File DSN). Системные источники данных доступны всем приложениям, работающим на данном компьютере, независимо от учетной записи, под которой они запущены. Пользовательские источники данных определены для каждой учетной записи. И наконец, файловые DSN хранятся в файлах, их может использовать любое приложение, в том числе выполняемое на других компьютерах при наличии доступа к соответствующему файлу DSN.

Примерный вид диалога настройки DSN приведен на рисунке 3.4.

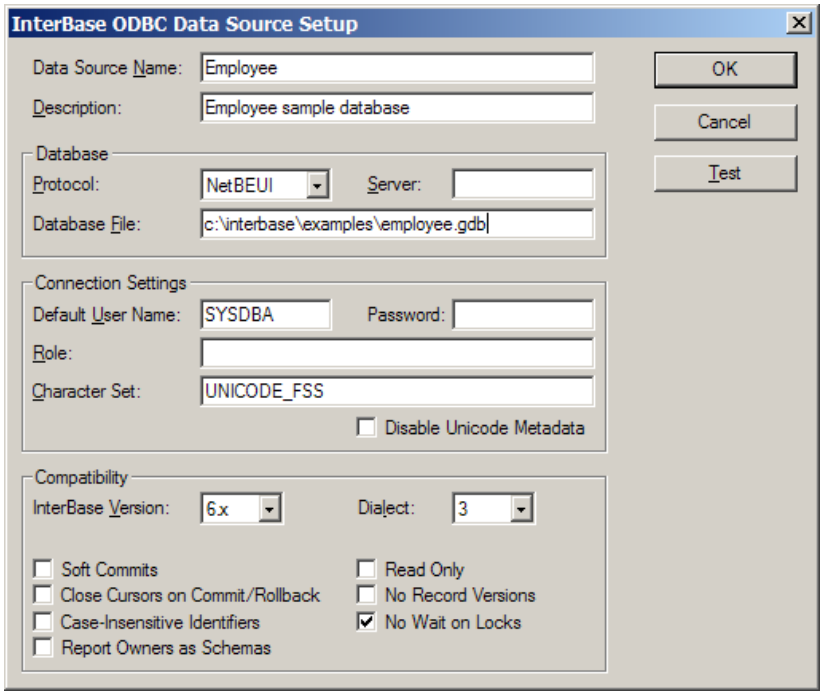


Рис. 3.4. Диалог настройки источника данных

В случае использования DSN-less-соединения приложение должно передать все параметры соединения вместе с именем драйвера в строке соединения.

Ниже перечислены все опции настройки источника данных параллельно для настройки DSN и при задании в строке соединения. Параметр Options содержит

битовую маску, каждый бит которой соответствует установке некоторого флага в диалоге DSN.

Таблица 3.1. Параметры настройки источника данных

Поле диалога настройки DSN	Параметр строки соединения	Значение по умолчанию	Описание
–	Driver	–	Имя ODBC-драйвера. Используется только для DSN-less-соединений. Если имя драйвера содержит пробелы, необходимо заключить его в фигурные скобки. Для Gemini ODBC-драйвера нужно задавать так: DRIVER={Gemini InterBase ODBC Driver 2.0};
Data Source Name	DSN	–	Имя источника данных. Используется для соединений с применением DSN
Protocol	Protocol	1	Протокол, используемый для соединения. Задается числовым кодом, возможные значения которого таковы: 1 – Local; 2 – TCP; 3 – NetBEUI (Named Pipes); 4 – SPX
Server	Server	""	Имя серверного компьютера для удаленных протоколов. Пустое имя соответствует локальному серверу
Database File	Database	–	Имя файла базы данных. Обязательный параметр
Default User Name	UID	""	Имя пользователя
Password	PWD	""	Пароль пользователя. Не рекомендуется задавать пароль в настройках DSN, поскольку он хранится в реестре в открытом виде
Role	Role	""	Имя роли SQL, используемое при подключении к базе данных
Characte	Charset	""	Название кодировки пользова-

Поле диалога настройки DSN	Параметр строки соединения	Значение по умолчанию	Описание
r Set			тельского подключения
InterBase Version	Version	6	Номер версии сервера, в котором была создана база данных
Dialect	Dialect	3	InterBase SQL-диалект
Soft Commits	Options	256 (0x100)	Использовать COMMIT RETAINING. Соответствует биту 8 (маска 256) поля Options. Внимание! Установленному биту соответствует "жесткий" COMMIT
Close Cursors on Commit			Закрывать курсоры при завершении транзакции. Соответствует биту 0 (маска 1) поля Options
Case-insensitive identifiers			Отключает поддержку идентификаторов, зависящих от регистра в диалекте 3. Соответствует биту 2 (маска 4) поля Options
Report Owners as Schemas			Выдавать имя владельца объектов при описании структуры базы данных. Для большинства приложений приводит к проблемам. Соответствует биту 1 (маска 2) поля Options
No Record Versions			Запрещает чтение старых версий записей в уровне изоляции READ COMMITTED. Соответствует бит 4 (маска 16) поля Options
No Wait On Locks			Запрещает ожидание транзакции в случае конфликтов обновлений. Соответствует биту 5 (маска 32) поля Options
Read Only	ReadOnly	0	Устанавливает режим обращения "только чтение" к базе данных

Рассмотрим несколько примеров задания строки соединения из приложения. В качестве приложения возьмем скрипт на языке VB Script. Для запуска теста вам необходимо выполнить скрипт с помощью команды cscript имя-файла.vbs.

DSN-less соединение

В этом примере все параметры соединения устанавливаются программно.

```
dim conn
set conn = CreateObject("ADODB.Connection")
conn.open "Driver={Gemini InterBase ODBC Driver
2.0};Protocol=2;Server=localhost;Database=z:\borland\InterBase\
examples\employee.gdb;Dialect=3;UID=sysdba;PWD=masterkey"
```

Соединение с использованием DSN

В этом примере вам необходимо создать источник данных с именем Employee. В программе передаются только имя пользователя и пароль.

```
dim conn
set conn = CreateObject("ADODB.Connection")
conn.open "DSN=Employee;UID=sysdba;PWD=masterkey"
```

Вероятные проблемы и способы их решения

Сообщение "Client library cannot be loaded" при попытке установить соединение. – На компьютере не установлен клиент InterBase/Firebird. Проверьте наличие библиотеки GDS32.DLL в системном каталоге Windows.

Ошибка "Optional feature not supported". – В строке соединения и настройке DSN имя сервера указано дважды, например, в поле Server и в поле Database. При этом на сервер приходит строка соединения следующего вида: Localhost:localhost:/path_to_database/db.gdb. Проверьте настройки DSN и строку коннекта.

Ошибка "Unavailable database" при соединении из службы NT, например IIS или MS OLAP Services. – Возможная причина – источник данных настроен на использование локального протокола. Локальный протокол InterBase/Firebird не поддерживает соединения из служб Windows NT/2000. Используйте протокол TCP или Named Pipes с пустым именем сервера.

При установке на "чистую" Windows NT 4.0 имя DSN в ODBC Data Source отображается неправильно. – На компьютере не установлен ODBC Driver Manager версии 3.5 или выше. Установите обновление MDAC, которое можно загрузить с Web-сайта Microsoft.

При записи пустой строки в связанную таблицу Microsoft Access выдается ошибка "You tried to assign the NULL variable to the variable that is not a Variant data type". – К сожалению, Microsoft Access не допускает хранения пустых строк в связанной таблице. Пустая строка автоматически преобразуются в NULL; если при этом поле таблицы объявлено как NOT NULL, возникает данная ошибка.

Невозможно обновить связанную таблицу в Microsoft Access, содержащую вычисляемые поля. – Клиентская библиотека InterBase не содержит средств для определения того, какие из полей результата выборки являются обновляемыми. При обновлении любого поля таблицы Microsoft Access выполняет команду обновления всех полей, включая вычисляемые, что приводит к ошибочной ситуации.

Заключение

Если вы применяете приложения, рассчитанные на использование источников данных ODBC, такие, как настольные офисные системы, языки программирования или даже серверы баз данных, использование ODBC, вероятно, будет самым простым и надежным способом интегрировать их с базами данных InterBase/Firebird. Для разработчиков ODBC предоставляет возможность проектировать приложения, способные однообразно работать с InterBase/Firebird наравне с другими серверами СУБД в качестве источника данных и при необходимости легко переносить приложения между разными типами серверов.

В этой главе описан драйвер Gemini ODBC, его настройка и использование для доступа к базам данных InterBase. Свежую информацию о драйвере, сообщения о выпуске новых версий вы всегда сможете найти на Web-сайте поддержки данной книги www.InterBase-world.com, а также на сайте Gemini www.ibdatabase.com.

Создание клиентов на Java. InterClient и JDBC

Технология Java является одной из самых бурно развивающихся в мире. Поэтому поддержка Java является необходимым условием существования любого сервера баз данных. Разумеется, и InterBase, и все его клоны поддерживают возможность работы с приложениями на Java.

Для того чтобы работать с базами данных InterBase в Java-приложениях, необходимо задействовать технологию JDBC (часто расшифровывается как Java DataBase Connectivity). Для подключения JDBC к InterBase можно использовать несколько способов.

Во-первых, можно применять мост JDBC–ODBC, который может использоваться в комбинации с многочисленными ODBC-драйверами для InterBase и его клонов – например, Gemini. Но сейчас разработчики на Java стараются не использовать мост ODBC–JDBC, так как это существенно увеличивает зависимость приложения от платформы. Для того чтобы разрабатывать независимые от платформы приложения, существует "сетевой" JDBC-драйвер для InterBase, так называемый InterClient.

InterClient относится к 3-му типу JDBC-драйверов согласно классификации компании Sun. InterClient состоит из двух частей – собственно драйвера, написанного на "чистом" Java, и промежуточного сервера (называемого InterServer), который транслирует вызовы JDBC-команд в команды InterBase. Данный способ является наиболее безопасным и гибким для работы в Интернете. Именно InterClient мы будем использовать в качестве основы для примера, в котором продемонстрируем разработку клиента InterBase на Java.

Все приведенные ниже примеры приводятся в предположении, что на рабочем компьютере, где будет установлен InterClient, предварительно установлен пакет JDK1.3 или JDK1.4. Работа InterClient 2.x с более ранними версиями JDK может вызвать некоторые проблемы и иногда необходимость использовать более раннюю версию InterClient 1.6.

Установка InterClient

В настоящий момент самой последней является версия InterClient 2.01. Надо отметить, что InterClient является бесплатным и его можно свободно скачать с сайта http://Firebird.sourceforge.net/ или с сайта поддержки данной книги www.InterBase-world.com. Существуют версии InterClient как для Windows 95/98/ Me/NT 4/2000, так и для Linux. Мы рассмотрим установку и работу под Windows, хотя и установка под Linux ненамного сложнее и вся установка сводится к распаковке tar-архива и запуску инсталляционных скриптов.

В скачанном архиве InterClient для Windows находится программа-установщик, которая производит все необходимые действия по установке нужных файлов и модификации реестра.

InterClient состоит из двух основных частей – драйвера для JDBC, находящегося в файле interclient.jar, и транслятора вызовов JDBC в вызовы InterBase –

программы InterServer. Для того чтобы можно было из программ на Java обращаться к серверу InterBase, функционирующему на каком-либо компьютере, необходимо, чтобы на этом же компьютере был запущен InterServer.

В ОС Windows 95/98/Me InterServer может функционировать только в режиме приложения, а в NT/2000 – и в режиме сервиса. В составе установочного пакета InterClient поставляется программа для конфигурирования InterServer, с помощью которой можно настроить параметры запуска InterServer и управлять его состоянием.

Далее, после того как отработала программа-установщик InterClient, нам необходимо сделать следующее.

Во-первых, проверить, записалась ли в файл services строка

```
interserver      3060/tcp              # InterBase InterServer
```

Этот файл находится в каталоге c:\windows\services в ОС Windows 95/98/Me и в каталоге c:\winnt\system32\drivers\etc\services в NT/2000. Данная запись необходима для того, чтобы к InterServer можно было обратиться по сети.

Во-вторых, необходимо запустить программу для конфигурирования InterServer – isconfig.exe и убедиться, что InterServer функционирует так, как вам требуется.

И в-третьих, необходимо добавить путь к файлу interclient.jar в JAVA CLASSPATH. Например, если установка InterClient была произведена в каталог C:\Program Files\Firebird\InterClient, то в JAVA CLASSPATH необходимо добавить C:\Program Files\Firebird\InterClient\interclient.jar.

На этом установка InterClient завершена и мы можем приступить к разработке приложения на Java. Однако сначала рассмотрим некоторые типичные проблемы и способы их разрешения, которые могут возникнуть в процессе работы.

Communication Diagnostics

Прежде чем приниматься писать приложение базы данных на Java, необходимо проверить, а доступна ли нужная база данных. Для проверки возможности соединения с базами данных InterBase через InterClient служит Java-апплет Communication Diagnostics, входящий в состав поставки InterClient. Чтобы его запустить, можно либо открыть файл CommDiag.html, после чего в отдельном окне запустится этот апплет (естественно, ваш браузер должен поддерживать JAVA-апплеты), либо запустить его в командной строке следующим образом:

```
Java InterBase.interclient.utils.CommDiag
```

При этом появится окно диалога для проверки соединения с базой данных.

Если у вас в настройках Windows стоят региональные установки (*Панель управления\Язык и стандарты, закладка Общие\Язык(Местоположение)*) для России – "Русский", то скорее всего вместо нормальных русских букв вы увидите непонятные символы, свидетельствующие о неправильной кодировке файлов ресурсов для русского языка.

Бороться с некорректным представлением сообщений InterClient на русском языке можно двумя способами: либо установить в качестве значения для "Язык(Местоположение)" язык "Английский" и получить таким образом окно диа-

гностики на английском языке, либо скачать исправленную версию JDBC-драйвера InterBase interclient.jar с корректной поддержкой русского языка. Его можно скачать с сайта http://people.comita.spb.ru/users/sergeya/Java/interclient.jar или с сайта поддержки этой книги www.InterBase-world.com.

Здесь приводится английский вариант окна диагностики, которое изображено на рисунке 3.5:

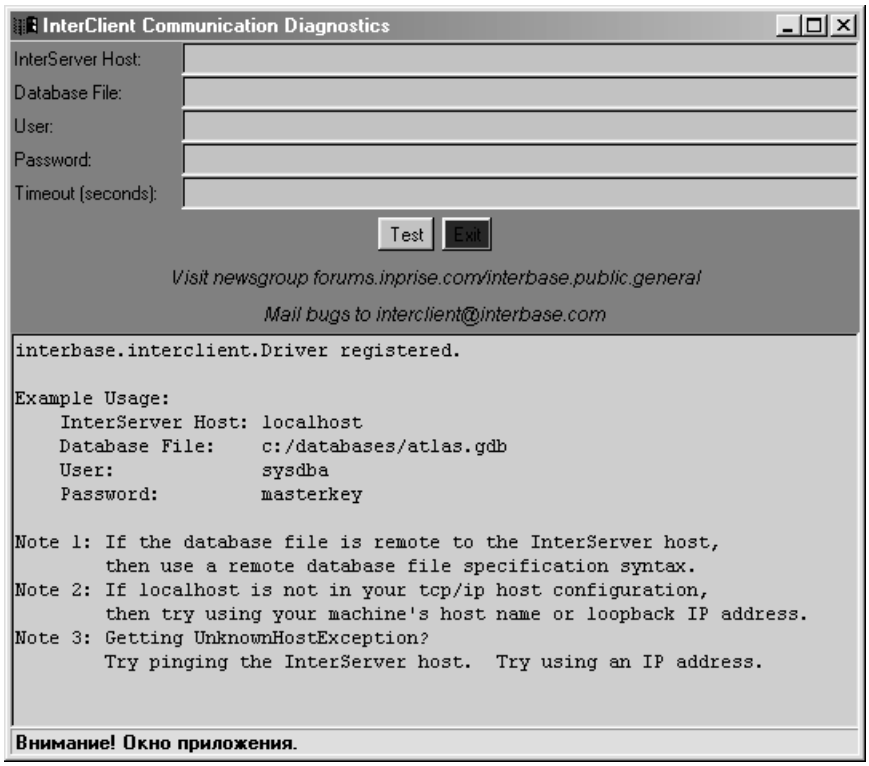


Рис. 3.5. Окно диагностики соединения через InterClient

Как видно из рисунка, для того, чтобы проверить соединение с базой данных через InterClient, необходимо заполнить поля "InterServer Host", "Database File", а также "User" и "Password". Если вы работаете по локальной сети (или вообще на собственной локальной машине), то поле "Timeout" можно не задавать.

Допустим, у нас есть база данных test.gdb под управлением InterBase на компьютере-сервере server_nt, к которой мы хотим подключиться через InterClient. Если обычная строка соединения к базе выглядит так:

```
server_nt:C:\Database\test.gdb
```

то необходимо ввести в поле "InterServerHost" значение "server_nt", а в поле Database File – "C:/Database/test.gdb". Обратите внимание, что в качестве разделителя каталогов используется прямая косая черта. Вообще говоря, InterClient позволяет использовать для разделения каталогов в строке соединения и привычную Windows-пользователям обратную косую черту, однако лучше исполь-

зовать прямую, чтобы не было необходимости квотировать обратную косую черту в программе на Java.

Нажимаем кнопку Test и видим результаты подсоединения к базе данных. Помните, что программа-транслятор InterServer обязательно должна функционировать на том же компьютере, где работает InterBase. Поэтому "InterServer Host" совпадает с именем сервера, на котором работает InterBase. Допускается, но не рекомендуется указывать в поле "InterServer Host" IP-адрес компьютера-сервера.

Итак, если соединение прошло успешно, то увидите успешное подтверждение соединения в окне диагностики. Пример сведений, выдаваемых в окне диагностики для базы данных на локальном компьютере со строкой соединения localhost:C:\Database\test.gdb, приведен ниже:

```
InterClient Release:                    2.0.1 Test Build, Cli-
ent/Server Edition
InterClient compatible JRE versions:    1.3
InterClient compatible IB versions:     5, 6
InterClient driver name:                Inter-
Base.interclient.Driver
InterClient JDBC protocol:              jdbc:InterBase:
InterClient JDBC protocol version:      20001
InterClient expiration date:            no expiration date

Testing database URL
jdbc:InterBase://localhost/C:/Database/test.gdb.
Connection established to
jdbc:InterBase://localhost/C:/Database/test.gdb

Database product name:      InterBase
Database product version:   WI-T6.2.773 Firebird 1.0
Database ODS version:       10.0
Database Page Size:         8,192 bytes
Database Page Allocation:   134 pages
Database Size:              1,072 Kbytes
Database SQL Dialect:       3

Middleware JDBC/Net server name:             InterServer
Middleware JDBC/Net server version:          2.0.1 Test Build
Middleware JDBC/Net server protocol version: 20001
Middleware JDBC/Net server expiration date:  no expiration date
Middleware JDBC/Net server port:             3060

Test connection closed.
***** NO Installation problems detected! *****
```

Помимо разнообразных сведений, в окне диагностики выводится строка JDBC-соединения, которая была использована для проверки связи с базой данных:

```
jdbc:InterBase://localhost/C:/Database/test.gdb
```

Это очень удобный способ формировать синтаксически правильные строки JDBC-соединения. Давайте сохраним эту строку для будущего использования в нашей программе.

Пример приложения на Java

Давайте рассмотрим простой пример приложения на Java, который будет устанавливать связь с базой данных, производить выборку данных из таблицы и распечатывать ее на экране.

Хотя в данном примере показывается работа с InterBase, можно заметить, что он похож на примеры работы с другими серверами СУБД. Это связано с тем, что Java, а точнее, JDBC предоставляет универсальный способ общения своих приложений и любых СУБД, для которых есть JDBC-драйвера. InterBase не является исключением, и любой Java-разработчик сможет легко разобраться в использовании JDBC-драйвера InterBase, если он ранее уже работал с JDBC.

Итак, вот пример программы, которая находится в файле SampleInterBase2JAVA.Java:

```
import Java.sql.*;

public class SampleInterBase2JAVA {
  public static void main(String[] args){
  // срока соединения с базой данных InterBase
  String url =
"jdbc:InterBase://localhost/C:/Database/test.gdb";
  try {
        // загружаем драйвер для InterBase
        Class.forName("InterBase.interclient.Driver");
     } catch(Java.lang.ClassNotFoundException e) {
        // в случае, если драйвер не найден,
        // выдаем сообщение об ошибке
       System.err.println(e.getMessage());
  }
  Connection conn = null; // соединение с базой данных
  try {
       // создаем соединение с базой данных (объект conn)
       // указанной в строке соединения url
       // используем пользователя/пароль:
SYSDBA/masterkey
     conn = DriverManager.getConnection(url,"SYSDBA",
"masterkey");
     }catch(Java.sql.SQLException sqle){
       // в случае проблем с подключением
       // выдаем соответствующее сообщение об ошибке
    System.err.println(sqle.getMessage());
  }
       //после создания соединения
       // создаем объект выражение - stmt
        Statement stmt = null;
  try{
       stmt = conn.createStatement();
    }catch(Java.sql.SQLException EsqlConn){
       System.err.println(EsqlConn.getMessage());
  }
```

```
       // текст SQL-запроса
  String sSQL = "Select ID, NAME  FROM TableExample";

      ResultSet rs = null;
  try{
       // выполняем запрос и помещаем результат
       // в объект ResultSet rs
      rs = stmt.executeQuery(sSQL);
      }catch(Java.sql.SQLException EsqlConn){
            System.err.println(EsqlConn.getMessage());
}
      // распечатываем результат на экране
  try{
       while (rs.next()) {
            int id = rs.getInt("ID");
            String s = rs.getString("NAME");
            System.out.println(id + "   " + s);
        }
       }catch(Java.sql.SQLException EsqlFetch){
            System.err.println(EsqlFetch.getMessage());
  }
}
}
```

Теперь можно попытаться откомпилировать и запустить эту тестовую программу следующим образом:

```
Java.exe   SampleInterBase2JAVA
```

Если все установлено правильно и существует такая база, то вы получите два столбца с результатами. Однако есть одна особенность, которую необходимо знать для работы из Java с базами данными, содержащими кириллицу.

Если вы следовали рекомендациям, приведенным в главе "Русификация InterBase" (ч. 1), то ваша тестовая база, в которой предполагается хранить русские символы, создана с использованием набора символа (charser) WIN1251.

Если попытаться получить из базы данных строки, содержащие русские символы описанным в приведенном выше примере способом, то в результате выборки будут находиться символы с некорректной кодировкой, которые прочитать будет невозможно. Чтобы заставить JDBC-драйвер InterBase использовать правильный набор символов для работы с кириллицей, необходимо указать его в параметрах соединения. Для этого следует создать объект Properties и поместить в него параметры соединения. За набор символов, который будет использоваться для соединения с базой данных, отвечает параметр charset. Вот пример соединения:

```
// задаем параметры соединения – строку соединения,
// имя пользователя, пароль и набор символов
String url = "jdbc:InterBase://localhost/C:/Database/test.gdb";
String uName = "SYSDBA";
String pass = "masterkey";
String charSet="cp1251";
// создаем структуру для хранения параметров соединения
```

```
Properties prop = new Properties();
prop.put("user", uName);
prop.put("password", pass);
prop.put("charset", charSet);
// получаем соединение с базой данных InterBase
// с указанием используемого набора символов
Connection db = DriverManager.getConnection(url, prop);
```

Таким образом разрешается проблема в приложениях Java с работой с кириллическими символами в базах данных InterBase.

Заключение

В этой главе описан JDBC-драйвер для InterBase и его клонов – InterClient и приведен небольшой пример приложения на Java, подключающегося к базе данных InterBase и выполняющего простейший запрос. Основная идея этой главы в том, чтобы показать, насколько легко связать InterBase и Java.

Часть 4
Администрирование и архитектура InterBase

Установка InterBase – взгляд изнутри

InterBase как встраиваемая СУБД

Материал этой главы будет посвящен углубленному рассмотрению процесса установки InterBase и его клонов на ОС Windows. В этой главе мы попытаемся понять, что значит определение "встроенная" (embedded) СУБД, которое так часто используют по отношению к InterBase.

> Почему изложение материала этой главы ориентировано на ОС Windows? Что бы ни говорили поклонники Linux, но наиболее значительный процент серверных инсталляций InterBase осуществляется именно под Windows, а что касается количества клиентских установок, то ОС Windows здесь вообще вне конкуренции. Поэтому мы рассмотрим вопросы встраивания InterBase именно в Windows-приложения.

Легковесность и простота администрирования делают InterBase идеальным кандидатом для создания тиражируемых программных систем, которые функционируют по принципу "установил и забыл". СУБД в таком приложении играет "закулисную" роль – в идеале пользователь не должен ничего знать о том, какая СУБД обслуживает его запросы. К встроенной СУБД предъявляются высокие требования по надежности и особые условия администрирования, сводящие к минимуму участие администратора СУБД.

Чтобы разобраться в сущности "встраивания" InterBase в приложения баз данных, необходимо более подробно изучить процессы, протекающие при обычной, штатной установке сервера и клиента InterBase. Разобравшись в сути процессов установки, легко будет перейти к созданию собственных установщиков InterBase, которые можно будет встроить в собственные программы.

> Все процессы установки рассматриваются на примере клона InterBase 6.x – Firebird 1.0.

Установка InterBase на платформе Windows

Один из установщиков InterBase, который предоставляет удобный графический интерфейс и поможет самому неискушенному пользователю справиться с процессом инсталляции, описан в главе "Установка InterBase" в самом начале этой книги. Однако сейчас нас интересует то, что происходит за кулисами удобных установщиков: какие файлы и куда копируются, какие ключи в реестре изменяются и добавляются, какие условия проверяются в процессе установки.

Важный вопрос, который обязательно возникнет в процессе чтения главы, откуда брать файлы для создания собственного установщика? Их можно взять и из стандартной установки InterBase, которая была проведена установщиком, а можно загрузить прямо с "родных" сайтов InterBase или Firebird.

Установка клиента под Windows

Давайте рассмотрим, что происходит при установке клиента под Windows. Каким бы скрытым ни был процесс инсталляции сервера, некоторые исходные данные задать все равно придется, явно запросив их у пользователя или установив какие-то по умолчанию значения.

Во время процесса установки InterBase-клиента нужно указать каталог, куда будет устанавливаться InterBase, – назовем его <InterBase_root>. Установка клиента включает следующие шаги:

1. Копирование файлов, входящих в состав клиента.
2. Регистрация файлов для совместного использования.
3. Создание реестровых ключей.
4. Регистрация сервиса TCP/IP.

Копирование файлов

Как описано ниже в главе "Состав модулей InterBase", минимальный корректный клиент InterBase состоит из трех файлов – gds32.dll, interbase.msg и msvcrt.dll.

> Опытные специалисты могут заявить, что абсолютный минимум – это библиотека gds32.dll, которую можно положить в тот же каталог, в котором находится и приложение. Однако для 100 %-ной гарантии правильной установки клиента необходимо все же копировать как минимум 3 файла.

Давайте рассмотрим назначение файлов в этом минималистском варианте установки, которого будет достаточно для работы большинства приложений.

В файле interbase.msg находятся тексты сообщений об ошибках сервера и клиента. Необходимо, чтобы этот файл имел ту же версию, которую имеет и библиотека gds32.dll. Этот файл устанавливается в установочный каталог InterBase.

Файл msvcrt.dll – это одна из самых банальных динамических библиотек, которая почти всегда имеется в системе Windows. InterBase 6.x требует, чтобы версия этой библиотеки была 5.00.7303 или старше. Обычно этот файл устанавливается в системный каталог Windows.

Самым важным файлом является динамическая библиотека gds32.dll, в которой сосредоточена вся основная функциональность, называемая нами "клиентом InterBase". Поэтому установке файла gds32.dll следует уделить особое внимание.

Прежде чем установить gds32.dll на компьютер, необходимо убедиться, что на компьютере нет другой копии этой динамической библиотеки. Для этого необходимо осуществить поиск этого файла в следующих каталогах: системном каталоге Windows (это Windows\System для 9x ОС и Winnt\System32 для NT/2000); в установочных каталогах InterBase 4.x, 5.x и 6.x; в установочном каталоге BDE, а также во всех каталогах, которые включены в переменную среды PATH.

Если будет найдена копия gds32.dll в одном из перечисленных каталогах, то необходимо выяснить ее версию и сравнить с версией gds32.dll, которую вы устанавливаете.

Если устанавливаемая gds32.dll имеет версию новее, чем у существующей библиотеки, то можно произвести замену старой версии на новую. При этом желательно предупредить пользователя о том, что совершается замена. Однако ни в коем случае нельзя заменять новую версию более старой! Это связано с тем, что новые версии библиотеки gds32.dll смогут взаимодействовать со "старыми" версиями сервера InterBase, но не наоборот!

Если решение об установке gds32.dll было принято, то рекомендуется поместить ее в системный каталог Windows.

Совместное использование gds32.dll, InterBase.msg и mscvrt.dll

Представьте ситуацию, когда на одном компьютере оказались два приложения, использующие клиент InterBase. Первое приложение успешно инсталлировалось, установив вместе с собой InterBase-клиента. Второе приложение в процессе установки обнаружило, что клиент InterBase, удовлетворяющий его, уже существует, и ничего не стало устанавливать. Затем первое приложение было удалено с компьютера с помощью автоматического установщика. Правила хорошего тона Windows требуют, чтобы удаляющееся приложение убирало за собой ненужные dll и другие вспомогательные файлы. Однако gds32.dll и остальные файлы все еще используется другим приложением! Чтобы предотвратить удаление еще нужного клиента InterBase, необходимо указать установщику, что библиотеки gds32.dll, msvcrt.dll и файл InterBase.msg используются еще одним приложением. Для этой цели служит специальный счетчик, хранящийся в реестре Windows. Вне зависимости от того, устанавливало ли приложение при своей установке какие-либо файлы dll или удовлетворилось существующими, необходимо при установке увеличить этот счетчик на единицу, а при удалении программного обеспечения – уменьшить. Когда значение счетчика примет значение 0, то это будет сигналом, что данную библиотеку можно удалить. Вот ветка реестра, где располагаются счетчики для совместно используемых динамических библиотек:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\
SharedDLLs
```

При установке клиента InterBase необходимо в этой ветке реестра искать ключ типа DWORD с именем C:\WINDOWS\System32\gds32.dll, и если его не существует, то необходимо создать его (это означает, что установка производится в первый раз) и присвоить ему значение 1. Если такой ключ уже существует, то нужно увеличить его значение на единицу. Точно такую операцию надо проделать над ключами, содержащими ссылки на msvcrt.dll и InterBase.msg.

Ключи в реестре для клиента InterBase

После установки клиента InterBase необходимо зарегистрировать его в Windows, чтобы дать другим приложениям возможность его использовать. Это делается путем записи определенных значений в реестре Windows. В таблице 4.1

представлены все значения ключей в реестре, которые необходимо установить после установки клиента.

Таблица 4.1. Реестровые ключи для установки InterBase

Ключ	Значение
HKEY_LOCAL_MACHINE\SOFTWARE\Borland\ InterBase\Current Version\Root Directory	Установочный каталог InterBase. Например: C:\Program Files\Firebird\
HKEY_LOCAL_MACHINE\SOFTWARE\Borland\ InterBase\Current Version\Version	Версия библиотеки gds32.dll. Например, версия может быть WI-T6.2.679 Firebird 1.0 Final Release

Если вам нужен установщик для инсталляции InterBase в рамках установки своего программного обеспечения, то можно либо создавать эти ключи самостоятельно, написав для этого специальную программу, либо воспользоваться утилитой instreg.exe из состава InterBase-сервера, которая пропишет эти значения в реестре.

Регистрация TCP/IP-сервиса при клиентской установке

Обычно клиент и сервер InterBase, будучи на разных компьютерах, связываются по протоколу TCP/IP. Чтобы получить возможность общаться по TCP/IP, необходимо серверу InterBase поставить в соответствие порт, по которому клиент будет общаться с сервером. Для этого надо в файл services добавить такую строку:

```
gds_db      3050/tcp      # InterBase
```

и после строки добавить перевод строки. Если такая строка уже присутствует в файле services, то ничего добавлять не следует – две одинаковые строки нежелательны.

Добавление этой строки в services означает, что для сетевого общения клиентов и сервера InterBase на данном компьютере выделен порт 3050. Такая запись должна быть на сервере и на всех клиентских компьютерах.

Надо отметить, что в Firebird 1.x и Yaffil 1.0 порт 3050 задан по умолчанию, можно ничего не прописывать в services. Единственное условие – система Windows, на которой устанавливается Firebird, должна поддерживать спецификацию winsock2. Это умеют делать все Windows, начиная с Windows 98, однако даже Windows 95 можно "научить", применив соответствующий пакет обновления.

Для всех предыдущих версий InterBase запись в services необходима для работы по TCP/IP.

Файл services в ОС Windows 9x находится в каталоге Windows, а в NT/2000/XP – в каталоге WINNT\System32\drivers\etc. Для его модификации в NT/2000/XP необходимо обладать правами администратора.

Как видите, установка клиента InterBase под Windows очень проста и не требует титанических усилий. Клиента InterBase легко включить в собственный инсталляционный пакет.

Установка InterBase-сервера на Windows

Собственный установщик тиражируемого приложения может установить сервер InterBase так же легко, как и клиента. Давайте рассмотрим необходимые для этого действия.

В процессе установки InterBase-сервера так же, как и при установке клиента, необходимо выбрать установочный каталог <InterBase_root>.

Далее, чтобы установить InterBase-сервер под Windows, необходимо действовать по следующему алгоритму:

1. Проверить, запущен ли InterBase-сервер на компьютере. Для этого необходимо вызвать функцию Windows API FindWindow и поискать процесс с именем "ibserver" или "ibremote". Если такой процесс найден, то необходимо прервать установку и сообщить пользователю, что он должен остановить любые версии InterBase, находящиеся на этом компьютере, прежде чем начинать инсталляцию.
2. Скопировать файлы сервера в предназначенные для них каталоги.
3. Зарегистрировать файлы для совместного использования.
4. Зарегистрировать сервис TCP/IP
5. Запустить InterBase-сервер.

Копирование файлов сервера

Здесь приведен пример для установки InterBase архитектуры SuperServer, как наиболее распространенный случай. При установке сервера копируются файлы, список которых и место назначения приведены в табл. 4.2.

Таблица 4.2. Файлы для установки InterBase-сервера

Файл	Описание файла	Куда копировать
ibserver.exe	Основной исполняемый файл InterBase	<InterBase_root>\Bin
ibconfig	Файл конфигурации InterBase	<InterBase_root>
isc4.gdb	База данных пользователей InterBase	<InterBase_root>
license.txt		<InterBase_root>
ib_license.dat		<InterBase_root>
gds32.dll	Клиентская библиотека	<InterBase_root>\Bin, системный каталог Windows
InterBase.msg	Файл сообщений	<InterBase_root>

Файл	Описание файла	Куда копировать
	InterBase	
Msvcrt.dll	Динамическая библиотека	Системный каталог Windows

Вы можете заметить, что последние 3 файла идентичны файлам, копируемым при клиентской установке. Единственное отличие – gds32.dll копируется также в каталог <InterBase_root>\Bin, вместе с основным исполняемым файлом ibserver.exe. Это служит дополнительной гарантией того, что ibserver.exe при запуске обнаружит gds32.dll той же самой версии, что и у него самого.

При копировании входящих в состав клиента файлов gds32.dll, InterBase.dll и msvcrt.dll, необходимо соблюдать те же самые условия, что и при клиентской установке.

Далее обязательно необходимо скопировать файлы ibserver.exe, ibconfig и isc4.gdb. Их назначение описано в главе "Состав модулей InterBase" этой части.

> Необходимо отметить, что в случае установки поверх уже существующего сервера ни в коем случае нельзя затирать существующую базу данных пользователей Isc4.gdb.

Также весьма важен вопрос о копировании файлов лицензионного соглашения и файлов с лицензиями. Если вы ставите бесплатную версию InterBase 6.x или его клона, то достаточно скопировать в <InterBase_root> файл с лицензионным соглашением InterBase Public License – LICENSE.TXT. Если же вы устанавливаете платную версию InterBase 6.x (например – InterBase 6.5), то также необходимо скопировать в <InterBase_root> файл лицензий ib_license.dat.

Совместное использование файлов

InterBase-сервер может быть встроен не только в ваше серверное программное обеспечение, поэтому основные файлы, входящие в состав его установки, надо зарегистрировать в Windows. Регистрация происходит точно так же, как описано выше в разделе "Совместное использование gds32.dll, InterBase.msg и mscvrt.dll", только помимо этих трех файлов надо зарегистрировать также файл ibserver.exe.

Ключи в реестре для сервера InterBase

При установке сервера, помимо записей о регистрации совместно используемых файлов, необходимо создать ключи в реестре. Например:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Borland\InterBase\CurrentVersion]
"Version"="WI-T6.2.679 Firebird Final Release 1.0"
"RootDirectory"="C:\\Program Files\\Firebird\\"
"GuardianOptions"="1"
"ServerDirectory"="C:\\Program Files\\Firebird\\bin"
"DefaultMode"="-r"
```

Параметры RootDirectory и Version уже знакомы нам – они описаны в установке клиента InterBase и имеют то же самое значение для сервера.

Параметр "DefaultMode"="-r" означает, что InterBase-сервер будет запускаться в режиме сервиса NT/2000/XP. Если нужно запускать InterBase в режиме приложения, то необходимо указать ключ "–a".

Ключ "GuardianOptions"="1" говорит о том, что необходимо запускать процесс-хранитель InterBase – ibguard.exe. Однако в рассматриваемый нами пример минимальной установки "гвардеец" не входит – и поэтому этот параметр может быть либо опущен, либо установлен в 0.

Параметр "ServerDirectory"="C:\\Program Files\\Firebird\\bin" указывает на каталог, из которого запускается ibserver.exe.

Регистрация TCP/IP-сервиса

Регистрация TCP/IP-сервиса при серверной установке ничем не отличается от регистрации при клиентской установке InterBase. Необходимо отметить, что процесс установки InterBase на NT/2000/XP должен выполняться с использованием учетной записи администратора, которая имеет права на модификацию файла services!

Запуск InterBase-сервера

InterBase-сервер, функционирующий под управлением NT/2000/XP, может выполняться в двух режимах – в виде службы (service) и в виде приложения. На Windows 9x InterBase может использоваться только в режиме приложения. Давайте рассмотрим, как настроить и запустить наш сервер после установки. Чтобы запустить InterBase в режиме приложения, необходимо выполнить команду "ibserver.exe –a". Чтобы установить InterBase в качестве сервиса на NT/2000/XP, следует воспользоваться утилитой instsvc.exe, которая зарегистрирует InterBase-сервер в качестве службы. Как известно, служба может либо запускаться автоматически при запуске Windows, либо запускаться вручную. Для запуска сервиса в автоматическом режиме необходимо выполнить команду

```
instsvc install "<InterBase_root>" -auto
```

Если InterBase устанавливается на систему Windows 2000/XP, то никаких дополнительных действий можно не предпринимать – эти системы сумеют самостоятельно перезапустить сервис, если он аварийно завершится. Другое дело – Windows NT4.0. Для перезапуска InterBase-сервера после возникновения сбоев необходимо воспользоваться специальной службой-хранителем – InterBase Guardian. В этом случае необходимо в процессе инсталляции скопировать файл ibguard.exe и установить его в качестве сервиса. В связи с тем, что поддержка NT4.0 в 2002 году прекращена, рассматривать установку этого сервиса здесь мы не будем, лишь порекомендуем InterBase 5.5 Embedded Installation Guide [15].

Расширенная установка InterBase-сервера

Рассмотренный выше минимум файлов не включает в себя часть расширенных возможностей InterBase. Чтобы воспользоваться многоязыковой поддержкой, стандартными UDF-функциями или своей библиотекой функций, необходимо скопировать еще ряд файлов, список которых приведен в табл. 4.3.

Таблица 4.3. Файлы для расширенной установки InterBase

Файл	Описание	Куда скопировать
ib_util.dll	Динамическая библиотека, содержащая необходимые для работы UDF-функции	<InterBase_root>\Bin
gdsintl.dll	Набор кодировок для поддержки национальных языков	<InterBase_root>\Bin
ib_udf.dll	Стандартная библиотека UDF-функций	<InterBase_root>\UDF

Если в тиражируемом приложении используются какие-либо собственные UDF-библиотеки, помимо ib_udf.dll, то их всех также необходимо скопировать в каталог UDF в установочном каталоге InterBase.

Пример установочного скрипта

Разумеется, существует множество приложений, использующих InterBase и его клоны в качестве встроенной СУБД, поэтому можно найти примеры готовых установочных скриптов, которые реализуют все вышеописанные действия по корректной установке сервера и клиента InterBase.

Пример установочного скрипта, который создает полноценный установщик сервера Firebird 1.0, можно скачать по адресу http://clientes.netvisao.pt/luiforra/ib/. Скрипт предназначен для использования в популярном бесплатном компиляторе установщиков InnoSetup (www.innosetup.com).

Резервное копирование базы данных и восстановление из резервной копии

Резервное копирование (backup) базы данных и *восстановление из резервной копии (restore)* – два важнейших и наиболее частых административных процесса, которые осуществляются разработчиками и администраторами InterBase.

Резервное копирование базы данных – практически единственный и самый надежный способ предохранить ваши данные от потери в результате поломки диска, сбоев электропитания, действий злоумышленников или ошибок программистов.

Помимо сохранения данных от возможных опасностей, в процессе резервного копирования создается независимый от платформы стабильный "снимок" базы данных, с помощью которого легко перенести данные на другую ОСили даже другую платформу. Также backup базы данных производит своего рода "освежение" данных в базе данных, производя сборку "мусора" во время процесса считывания данных. Полный цикл: резервное копирование и восстановление из резервной копии (часто эту последовательность действий называют *backup/restore* или сокращенно *b/r*) – является средством от излишнего "разбухания" базы данных, служит для корректировки статистической информации и является обязательным участником всех профилактических и "лечебных" процедур обслуживания базы данных. В процессе backup/restore сначала все данные из базы данных копируются в backup-копию базы данных – файл специального формата (не GDB), а затем на основе сохраненных данных база данных полностью пересоздается.

Можно сказать, что процессы backup и restore – лучшие друзья разработчика и администратора InterBase. Регулярное выполнение backup/restore поддерживает "здоровье" базы данных в прекрасном состоянии, предохраняя данные от порчи и возможных ошибок.

Помимо "лечебных" целей, процесс backup/restore дает возможность изменить многие ключевые характеристики базы данных – например, сменить размер страницы базы данных, установить для базы данных режим "только чтение" (read-only), разбить файл базы данных на несколько частей. Например, смена основных версий ODS (On-Disk Structure) и миграция от одной версии InterBase-сервера к другой происходят только при помощи процесса backup/restore.

Резервное копирование базы данных InterBase

> Резервное копирование (backup) базы данных – это процесс считывания всех данных из базы данных и сохранения их в виде одного или нескольких файлов на диске или устройстве резервного копирования. Во время backup происходит считывание самых последних версий записей в таблицах базы данных на момент запуска процесса резервного копирования. Старые версии записей никогда не попадают в backup.

Backup – это не просто сохранение базы данных путем простого файлового копирования средствами ОС. Во время backup специальная программа-клиент подключается к базе данных в режиме чтения и считывает все системные и пользова-

тельские данные в файл особого формата, который обычно имеет расширение gbk (groton backup, по всей видимости).

Для экономии дискового пространства часто бывает удобно упаковывать файл резервной копии с помощью архиватора (обычно ZIP), так как это позволяет сжать размер backup в 8–10 раз.

Памятуя о том, что backup базы данных InterBase – это обычное считывание информации из базы данных, выполняемое специальной программой в режиме клиентского доступа, перечислим следующие особенности резервного копирования:

- Backup базы данных InterBase может осуществляться одновременно с работой обычных клиентских программ.
- Backup базы данных InterBase содержит данные, которые находились в базе данных на момент начала подключения программы, осуществляющей резервное копирование. Все изменения, проводимые выполняющимися параллельно процессу резервного копирования клиентскими программами, в резервную копию не попадут.
- Во время backup происходит считывание каждой записи из каждой таблицы в базе данных. Таким образом, происходит сборка "мусора" в базе данных (garbage collection): версии записей или их фрагменты, которые не являются актуальными, уничтожаются. Место из-под удаленных версий освобождается, и оставшиеся данные переупаковываются. Подробнее о версиях записей и сборке "мусора" смотрите главу "Транзакции. Параметры транзакций" (ч. 1).
- В процессе резервного копирования во время посещения всех записей происходит пересчет статистики по индексам, что улучшает производительность операций, которые используют эти индексы.

Осуществить резервное копирование базы данных InterBase можно двумя основными способами – с помощью специальной утилиты gbak из комплекта поставки InterBase или воспользовавшись Services API (на тех версиях сервера, которые поддерживают это API). Помимо этих двух штатных способов, можно также написать собственную программу, которая будет производить backup/restore в каком-либо формате, однако этот трудоемкий способ мы рассматривать не будем, так как штатные средства удовлетворяют потребностям почти всех пользователей InterBase.

Инструмент командной строки gbak

Наиболее универсальным инструментом, позволяющим осуществить резервное копирование базы данных на любой платформе, является *gbak* – утилита командной строки, входящая в поставку InterBase. С помощью gbak можно обратиться к любому функционирующему InterBase-серверу и произвести считывание данных и получение на их основе резервной копии, а также восстановить базу данных из резервной копии.

Надо заметить, что в случае использования для работы баз данных, чьи версии не совпадают с версией gbak, действует принцип "обратной совместимости". Это значит, что более "старшие" версии gbak могут работать с серверами и базами данных, созданными и функционирующими под управлением "младших" версий

(например, gbak от 5.x может сделать backup базы данных, которая была создана в 4.x). Однако gbak от 4.x не сможет работать с базами данных, которые созданы в 5.x и старше, и также не сумеет распаковать резервные копии от старших версий. Единственным исключением является случай, когда gbak запускается под управлением "младшего" сервера (например, 5.x), а в качестве источника данных указывает старший сервер (например, 6.x), в этом случае произойдет резервное копирование "наоборот" – данные из формата базы данных старшей версии попадут в формат backup младшей версии. Такой прием применяется при переходе от старших версий сервера на младшие – этот процесс называется "обратной миграцией" (подробности см. ниже в главе "Миграция").

Давайте рассмотрим утилиту gbak поподробнее. Для того чтобы создать резервную копию базы данных, необходимо воспользоваться следующим образцом запуска gbak:

```
gbak [-B] [options] <база_данных-источник> <файл резервной копии>
```

Переключатель –B означает, что необходимо выполнить резервное копирование базы данных, путь к которой указан как <база_данных-источник>, а результаты резервного копирования упаковать в файл, указанный как <файл резервной копии>. Обратите внимание, что –B взято в квадратные скобки. По общепринятому соглашению квадратные скобки означают, что параметр внутри них необязателен. В нашем случае это значит, что gbak без параметров будет делать именно backup базы данных.

Особый интерес представляют опции, с помощью которых можно управлять процессом резервного копирования. Набор дополнительных ключей (опций), представленный предложением [options], описан в таблице 4.4, взятой из [4].

Таблица 4.4. Опции gbak, применяемые при создании резервной копии

Опция	Описание
–b[ackup_database]	Осуществить резервное копирование базы данных
Опции, влияющие на процесс создания резервной копии	
–co[nvert]	Преобразовать внешние файлы во внутренние таблицы
–e[xpand]	Не производить сжатие резервной копии
–fa[ctor] n	Использовать блокирующий фактор n для ленточного накопителя
–g[arbage_collect]	Не собирать "мусор" во время резервного копирования
–ig[nore]	Игнорировать контрольные суммы
–l[imbo]	Игнорировать "зависшие" двухфазные транзакции (limbo)
–m[etadata]	Произвести резервное копирование только метаданных

Опция	Описание
–nt	Создать резервную копию в нетранспортабельном формате
–ol[d_descriptions]	Производить резервное копирование метаданных в формате "старого стиля", т.е. в режиме совместимости со старыми базами данных
–pas[sword]text	Пароль пользователя, подключающегося к базе данных для резервного копирования
–role name	Подсоединиться с использованием роли name
–se[rvice] servicename	Создать резервную копию на том же компьютере, где находится "база данных-источник". Для этого вызывается Service Manager на компьютере-сервере, причем формат вызова отличается для различных сетевых протоколов: TCP/IP hostname:service_mgr; SPX hostname@service_mgr; Named pipes \\hostname\service_mgr; Local service_mgr
–t[ransportable]	Создавать транспортабельную (переносимую) резервную копию – этот параметр включен по умолчанию
–u[ser] name	Имя пользователя, который подключается к базе данных для резервного копирования
–v[erbose]	Включить показ подробного протокола действий gbak во время backup
–y [file | suppress_output]	Направлять сообщения в файл (файла с таким именем не должно существовать) или подавить вывод сообщений
–z	Показать версию gbak и версию ядра InterBase-сервера

Давайте рассмотрим основные ключи, влияющие на процесс резервного копирования.

Во-первых, это ключи –t и –nt, которые определяют, является ли создаваемая резервная копия транспортабельной, т. е. переносимой с одной платформы на другую. По умолчанию (т. е. если не указывать ничего) создается транспортабельный backup, как при использовании ключа –t.

Во-вторых, это ключ –ig[nore], появление которого заставляет gbak не проверять контрольные суммы страниц базы данных, в результате чего в резервную копию могут попасть поврежденные страницы. Если этого ключа нет, то gbak при обнаружении страницы с запорченной контрольной суммой прекратит резервное копиро-

вание и выдаст соответствующую ошибку. Обычно ключ –ignore используют, когда производят починку базы данных (см. ниже главу "Починка базы данных").

В-третьих, переключатель –g[arbage_collect], который отключает сборку "мусора" во время резервного копирования. Как известно, InterBase хранит версии записей, измененных различными транзакциями. Это приводит к тому, что на страницах данных накапливается "мусор" – записи старых версий, которые никому не нужны. "Мусор" старых версий собирается, когда производится чтение самой "свежей", актуальной версии записи (подробнее о версиях и сборке "мусора" см. главу "Транзакции. Параметры транзакций" (ч. 1)). Так как резервное копирование – это чтение всех данных в базе данных, которое считывает каждую запись в каждой таблице, то backup также является инициатором крупномасштабного "субботника" – сборки "мусора" по всей базе данных. Надо сказать, что сборка "мусора" является хоть и полезным, но достаточно ресурсоемким процессом. Отключение сборки "мусора" приводит к значительному ускорению процесса резервного копирования. Это бывает исключительно полезным в случае очень больших (многогигабайтовых) баз данных. Однако в общем случае не рекомендуется отключать сборку "мусора" во время резервного копирования, за исключением случаев, связанных с починкой баз данных (см. главу "Починка базы данных").

Вкратце упомянем случаи использования других переключателей. Если вам необходимо получить резервную копию пустой базы данных, т. е. только ее метаданных, то воспользуйтесь переключателем –m[etadata]. Если вы используете несколько различных баз данных InterBase и применяете механизм двухфазного подтверждения транзакций, то транзакции, совершаемые сразу в двух базах данных, могут "зависнуть", т. е. получить промежуточное состояние – ни подтвержденное, ни отмененное, так называемое "in-limbo"-состояние (подробнее о транзакциях смотрите главу "Транзакции. Параметры транзакций"). Чтобы игнорировать результаты работы limbo-транзакций, т.е. версии записей, созданные этими транзакциями, применяется опция –l[imbo]. Чтобы в файл резервной копии попали данные, которые хранятся во внешних таблицах (external tables), используйте переключатель –co[nvert], который преобразует внешние файлы во внутренние таблицы и сделает их резервную копию.

Права для выполнения резервного копирования

Вопрос о правах, необходимых для резервного копирования базы данных – очень важный вопрос. Под "правами" имеются в виду самые различные привилегии, которые мы сейчас рассмотрим. Во-первых, это права на уровне InterBase. Только системный администратор или владелец базы может производить резервное копирование базы данных, т.е. "user name" всегда должно быть или SYSDBA, или именем пользователя-владельца. И пароль в опции password соответственно (по умолчанию для SYSDBA пароль задан строкой "masterkey") должен принадлежать данному пользователю.

Во-вторых, пользователь должен обладать правами на уровне ОС для осуществления процесса резервного копирования. В случае если компьютер-сервер, на котором функционирует InterBase, работает под управлением Windows NT/2000/XP, то пользователь, права которого серверный процесс InterBase, должен иметь привилегии для чтения и записи файла базы данных. Это совершенно

необходимое требование, обычно оно выполняется, так как по умолчанию серверный процесс InterBase пользуется правами доступа системной учетной записи (пользователь "Система"), которая по умолчанию обладает правами доступа ко всему диску.

В-третьих, необходимо отрегулировать права доступа на уровне ОС для пользователей, обращающихся с клиентских компьютеров к InterBase на компьютере-сервере. Если соединение происходит по протоколу TCP/IP, то никаких привилегий для работы с базой данных пользователю, работающему на компьютере-клиенте, давать не надо. Более того, к InterBase-серверу под управлением NT/2000/XP может обращаться любой пользователь, в том числе и не имеющий никаких прав, потому что при соединении по TCP/IP не производится проверки привилегий, специфичных для Windows. Если соединение происходит по протоколу Named Pipes, то пользователь должен иметь права на модификацию каталога и файла базы данных.

Помните, что при соединении по TCP/IP строка соединения имеет вид server_name: <диск>\<путь_на_диске_сервера_к_база данных>, а при соединении по протоколу Named Pipes – \\server_name\<диск>\<путь_на_диске_сервера_к_база данных>.

Необходимо также обратить ваше внимание на вопрос, связанный с безопасностью совместно используемых ресурсов – так называемых shared folders. Вам не нужно давать никакие права на такие ресурсы при соединении по любому протоколу – для работы с InterBase они совершенно не нужны.

В случае если компьютер-сервер с InterBase работает под управлением Linux, то для выполнения gbak также необходимо, чтобы он работал под учетной записью, имеющей права на модификацию файла базы данных. Также необходимо, чтобы пользователь, запускающий gbak, имел право запускать libgds.so – динамическую библиотеку, которая используется gbak для обращения к InterBase.

Резервное копирование многофайловых баз данных

Хотя база данных InterBase 6.x может иметь размер до 90 Тбайт, однако размер одного файла обычно ограничен размером 2 Гбайт (Firebird 1.x, а также InterBase начиная с 6.5 на NT/2000/XP под NTFS поддерживают файлы размером до 32 Гбайт). Поэтому необходимо коснуться вопроса о том, как осуществить резервное копирование базы данных InterBase, содержащей несколько файлов. Формат запуска утилиты gbak для резервного копирования многофайловой база данных следующий:

```
gbak [-B] [options] <база_данных-источник>  <файл резервной
копии1> size1[k|m|g] <файл резервной копии2> [ size2[k|m|g]
```

Как видно, формат команды аналогичен обычному однофайловому backup. Параметр <база_данных-источник> определяет путь к первому файлу базы данных. Информация об остальных файлах базы данных и путь к ним хранится в заголовке первого файла. Параметр <файл резервной копии1> определяет имя первого файла резервной копии, <файл резервной копии2> – имя второго файла резервной копии и т. д. Так как файлы backup также подчиняются ограничению в 2 (или 4) Гбайт, то при большом размере базы данных необходимо создавать многофайловую резервную копию. Каждому файлу резервной копии поставлен в соответствие пара-

метр "size", определяющий размер этого файла. По умолчанию размер файла указывается в байтах, однако суффикс, следующий сразу за размером, позволяет изменить единицы измерения – k (килобайты), m (мегабайты), g (гигабайты).

Для наглядности рассмотрим пример резервного копирования многофайловой базы данных в многофайловую резервную копию. Вот примерная команда для осуществления backup на сервере под управлением Windows:

```
gbak –b – user SYSDBA –password masterkey C:\database\verybigdb.gdb
D:\backups\dbback1.gbk 650M D:\backups\dbback2.gbk 650M
D:\backups\dbback3.gbk 650M
```

В этом примере база данных, первый файл которой назван verybigdb.gdb, будет упакована в 3 backup-файла размером по 650 Мбайт – например, для записи на 3 компакт-диска (которые обычно имеют размер 650 Мбайт). Если резервная копия получится меньше, то последний файл соответственно уменьшится (а быть может, и вообще не создастся).

Для резервного копирования многофайловых баз данных удобно пользоваться возможностью запуска gbak как сервиса на сервере базы данных, чтобы ускорить процесс backup за счет использования мощностей компьютера-сервера и отсутствия обмена по сети. Рассмотрим пример команды резервного копирования для выполнения gbak в качестве сервиса (пример для компьютера-сервера server_nt под управлением Windows):

```
gbak -b -user SYSDBA -password -service server_nt:service_mgr
C:\database\verybigdb.gdb D:\backups\dbback1.gbk 650M
D:\backups\dbback2.gbk 650M D:\backups\dbback1.gbk 650M
```

Остальные опции для резервного копирования многофайловых база данных аналогичны параметрам, описанным в таблице 4.4. Примеры использования всех опций, в том числе для резервного копирования на платформе *nix, можно найти в [4, гл. 7].

Резервное копирование при работе InterBase в режиме 24x7

Если условия работы InterBase таковы, что база данных имеет очень большой объем (порядка нескольких гигабайтов) и должна работать 7 дней в неделю и 24 часа в сутки (часто это называют режимом 24x7), например если InterBase обслуживает Web-сервер, который не должен простаивать, то существует особая методика получения резервной копии. Для этого используется механизм SHADOW-копий базы данных. В режиме использования SHADOW сервер осуществляет одновременную запись и чтение всех изменений из двух баз данных: основной и SHADOW-копии.

Когда нам необходимо сделать резервную копию, мы на 1–2 минуты останавливаем InterBase, переименовываем файл SHADOW-копии и снова запускаем сервер. Затем производим резервное копирование файла SHADOW (на другом компьютере). Этот подход позволяет избежать выполнения резервного копирования рабочей базы и, таким образом, чрезмерной загрузки компьютера-сервера.

Надо отметить, что целесообразность данного подхода прямо зависит от размера базы данных. Если база данных имеет размер в 2–3 Гбайта, не надо использовать описанный механизм – обычный backup будет приемлемым решением.

Другие инструменты для осуществления резервного копирования

Помимо универсальной утилиты командной строки gbak, множество других инструментов предоставляют удобный графический интерфейс для операций резервного копирования и восстановления из резервной копии. В документации по InterBase 6 приводятся примеры выполнения этих операций с использованием программы IBConsole, которая обычно входит в поставку InterBase и его клонов, однако лучше использовать инструменты из рекомендованного списка (см. приложение "Инструменты администратора и разработчика InterBase").

Помимо однократного осуществления backup часто возникает задача наладить регулярный процесс резервного копирования – например, ежедневный или даже чаще. Как автоматически наладить этот процесс? Для этого можно воспользоваться либо встроенными средствами ОС для организации регулярного копирования, т. е. с помощью штатного планировщика задач в определенное время запускать пакетный файл, содержащий команды для осуществления backup, либо использовать специальную программу-планировщик. Более удобным представляется второй способ – использование специальной программы. Для ОС Windows можно порекомендовать утилиту GBAK Sheduler (www.gbaksheduler.com), которая предоставляет удобный интерфейс для организации регулярного резервного копирования и совершенно бесплатна.

Восстановление из резервной копии

> Восстановление из резервной копии (restore) – это процесс создания базы данных на основе информации, извлекаемой из файла резервной копии.

В сущности, restore представляет собой создание пустой базы данных с заданными параметрами (размером страницы, режимом записи и т. д.). Затем в эту базу данных добавляются метаданные – таблицы, различные ограничения и проверки, триггеры, хранимые процедуры и т. д. Созданная база данных наполняется данными из файла резервной копии, после чего создаются необходимые индексы.

Пересоздание базы данных позволяет улучшить скорость работы, а также уменьшить размер базы данных за счет избавления от старых версий записей (ведь в backup включаются только актуальные версии записей) и переупаковки данных. Пересоздание индексов также улучшает скорость доступа к информации в базе данных. Обычно рекомендуется проводить полный цикл backup/restore не реже чем раз в месяц, чтобы избежать излишнего "разбухания" базы данных от накопленных версий.

Трудно переоценить возможность полного пересоздания базы данных (можно даже сказать – возрождения) на основе "мгновенного снимка" информации из базы данных. Только во время восстановления можно сменить основную версию ODS, перейти на другую платформу или ОС. Также restore является обязательным участником процесса преобразования однофайловой базы данных в многофайловую.

Восстановление с использованием инструмента gbak

Так же как и резервное копирование, восстановление можно осуществить двумя способами – с помощью утилиты gbak и с помощью Services API (если версия

InterBase-сервера имеет это API). Наиболее универсальным способом, который мы и рассмотрим, является использование gbak.

Формат команды восстановления база данных следующий:

```
gbak {-C|-R} [options] <файл_резервной_копии_источник> <файл
создаваемой базы данных>
```

При восстановлении с помощью gbak необходимо указать либо параметр –C, либо параметр –R, чтобы производить именно restore (по умолчанию gbak будет пытаться произвести backup). Параметр –C означает, что будет создан новый файл базы данных, но если его имя совпадет с уже существующим, то процесс будет остановлен с ошибкой, сигнализирующей о том, что файл с именем <файл создаваемой базы данных> уже существует. Параметр –R также приводит к созданию базы данных, но в случае совпадения имен без дополнительных вопросов перезапишет существующий файл базы данных.

Опции options, применяющиеся для того, чтобы повлиять на процесс восстановления, описаны в таблице 4.5.

Таблица 4.5. Опции gbak при восстановлении из резервной копии

Опция	Описание
–c[reate_database]	Восстанавливает базу данных из резервной копии
–bu[ffers]	Устанавливает размер буфера базы данных
–i[nactive]	Делает индексы неактивными после восстановления
–k[ill]	Не создает shadow-копий, которые были определены для базы данных ранее
–mo[de] [read_write | read_only}	Устанавливает режим записи для восстанавливаемой базы данных. Возможны значения read_write ("чтение и запись" – режим по умолчанию) и read_only ("только для чтения")
–n[o_validity]	Удаляет ограничения ссылочной целостности из восстанавливаемой базы данных, что позволяет восстанавливать те данные, которые не удовлетворяют этим ограничениям
–o[ne_at_a_time]	Восстанавливает одну таблицу зараз – это бывает полезным для частичного восстановления базы данных, которая содержит поврежденные данные
–p[age_size] n	Устанавливает размер страницы восстанавливаемой базы данных в n байт. Доступны значения 1024, 2048, 4196 или 8192; По умолчанию размер страницы – 1024 байта
–pas[sword] text	Пароль пользователя, подключающегося к базе данных для восстановление из резервной копии
–r[eplace_database]	Восстанавливать базу данных в новый файл, а если такой файл уже существует, то перезапи-

Опция	Описание
	сать поверх
–se[rvice] servicename •	Восстановить базу данных на том же компьютере, где находится база данных-оригинал. Для этого вызывается Service Manager на компьютере-сервере, причем формат вызова отличается для различных сетевых протоколов: TCP/IP hostname:service_mgr; SPX hostname@service_mgr; Named pipes \\hostname\service_mgr; Local service_mgr
–u[ser] name	Имя пользователя, который подключается к базе данных для восстановлениях
–use_[all_space]	Восстанавливает базу данных со 100 %-ным заполнением каждой страницы данных, вместо 80 %-ного заполненния по умолчанию
-v[erbose]	Включить показ подробного протокола действий gbak во время restore
–y [file | suppress_output]	Направлять сообщения в файл (файла с таким именем не должно существовать) или подавить вывод сообщений
–z	Показать версию gbak и версию ядра InterBase-сервера

Коротко рассмотрим некоторые важные ключи процесса восстановления. Во-первых, ключ -p[age_size] n, который устанавливает размер страницы создаваемой базы данных. Выполнить восстановление с этой опцией – это единственный способ изменить размер страницы базы данных.

Во-вторых, сочетание ключей -use_[all_space] и -mo[de] read_only позволяет создать базу данных только для чтения, с максимальным заполнением страниц данных. Это полезно при создании баз данных-справочников, распространяемых на компакт-дисках.

В-третьих, ключи -i[nactive] (деактивация индексов) и -n[o_validity] (удаление ограничений ссылочной целостности) часто применяются при восстановлении поврежденных баз данных (см. ниже главу "Починка базы данных" (ч. 4)).

Восстановление из резервных копий многофайловых баз данных

Из-за ограничения на размер одного файла базы данных в 2 (иногда 4) Гбайт, базы данных большего размера размещаются в нескольких файлах (так же как и резервные копии). Для восстановления многофайловой базы данных из многофайлового backup следует воспользоваться командой

```
gbak {-C|-R} [options]  <файл_резервной_копии1>
<файл_резервной_копии2> [<файл_резервной_копии3 …] <файл создава-
емой базы данных1> <размер базы данных1> <файл создаваемой базы
```

```
данных2>  <размер базы данных2> [<файл создаваемой базы данных3>
<размер базы данных3> ..]
```

Здесь <файл_резервной_копии N> – это N-й файл резервной копии. Восстановление будет производиться начиная с <файл_резервной_копии1>, затем будет обработан <файл_резервной_копии2>. База данных будет содержать несколько файлов начиная с <файл создаваемой БД1>, затем <файл создаваемой БД2> и т. д. После имени файла базы данных идет размер этого файла. Обратите внимание, что размер файлов исчисляется в страницах! Поэтому необходимо следить за тем, чтобы размер файла не вышел за обозначенные пределы в 2(4) Гбайт. Минимальный размер восстанавливаемого файла базы данных составляет 200 страниц. Также следует помнить о том, что не надо указывать размер последнего файла, – он увеличивается автоматически, чтобы вместить все данные из backup.

Как и в случае с резервным копированием больших многофайловых баз данных для ускорения процесса восстановления лучше всего воспользоваться запуском gbak в режиме сервиса.

Владелец базы данных

Резервное копирование базы данных может быть выполнено либо владельцем базы данных (owner), либо системным администратором (SYSDBA). Но восстановление базы данных может быть выполнено любым пользователем (исключая ситуацию, когда нужно восстановить базу данных поверх существующего файла, – это может осуществить только системный администратор SYSDBA или владелец). Восстановленная база данных принадлежит пользователю, который осуществил процесс восстановления т. е., он становится владельцем (owner) базы данных. Таким образом, процесс backup/restore является средством смены владельца базы данных.

Заключение

Процесс backup/restore дает возможность обновить вашу базу данных, очистить ее от "пепла" старых версий и пересоздать индексы. Чтобы уберечь данные от потери, необходимо регулярное резервное копирование, а для нормального функционирования базы данных надо периодически производить полное пересоздание базы данных с помощью восстановления ее из резервной копии.

Миграция

Под миграцией базы данных InterBase понимается несколько разных вещей – это и перенос существующей базы данных между различными версиями InterBase (например, с 4.x на 6.x), и смена ОС, под управлением которой функционирует сервер, и смена диалекта базы данных. Мы рассмотрим в этой главе все эти типы миграции и предоставим рекомендации по их безопасному осуществлению. Также будут рассмотрены варианты отката (обратной миграции) с новых версий InterBase на предыдущие.

Почему необходима миграция

Собственно, почему? Если вы любите эксперименты, то наверняка пробовали подключаться к базе данных от InterBase 5.x с помощью какого-нибудь клона 6.x и заметили, "старые" базы данных могут работать под управлением новых версий InterBase. Действительно, между версиями серверов существует совместимость "снизу вверх", когда старшая версия (например, 6.x) умеет работать с базами данных, созданных в предыдущей версии (например, 5.x), однако возможности такого взаимодействия ограничены. Например, нет совместимости при переносе базы данных между различными ОС и платформами – попробуйте скопировать файл базы данных с компьютера, на котором работает InterBase под Linux, на компьютер, где установлена Windows и соответствующая версия InterBase под Windows, а затем попытайтесь подключиться к этому файлу. База данных будет в большинстве случаев повреждена (не проводите таких экспериментов над рабочими базами данных!). Также не получится просто скопировать базу данных с системы на базе платформы Intel на систему SPARC и наоборот.

Дело в том, что база данных, созданная с использованием определенной версии сервера InterBase, который выполняется под управлением какой-либо ОС, имеет в своей структуре привязки к версии InterBase -сервера и к ОС.

База данных имеет различные версии ODS (On-disk structure) в зависимости от того, с помощью какой версии InterBase она была создана. ODS определяет, какова внутренняя структура файлов базы данных InterBase (подробнее об ODS см. главу "Структура базы данных InterBase"). При переходе от одной версии сервера к другой ODS может меняться, при этом включаются дополнительные возможности, которые задействованы в новых версиях InterBase. Хотя новые версии InterBase ради совместимости позволяют работать с ODS от предыдущих версий, но при этом новые возможности будут недоступны.

Чтобы использовать возможности, предоставляемые новыми версиями InterBase, необходимо обязательно осуществить миграцию базы данных, созданной в предыдущей версии, к соответствующей ODS.

При переходе от одной ОС к другой возникает более очевидная ситуация: либо InterBase просто откажется работать с базой данных, пришедшей (читай – переписанной) с другой ОС, либо выдаст множество ошибок, связанных с несовместимостью представления данных в разных ОС. Дело в том, что в каждой ОС существует собственная реализация некоторых типов данных и при попытке работать с базой данных на другой платформе неверно интерпретируются

значения, хранящиеся в базой данных. Миграция просто необходима при смене ОС, если вы желаете сохранить свою базу данных в целости и сохранности.

Аналогичная ситуация складывается при переходе с одной аппаратной платформы на другую – например, при переходе Intel->Sparc. Миграция необходима для корректной модификации тех данных в базе данных, которые зависят от аппаратной платформы.

Сущность процесса миграции

Миграция – это перенос баз данных между различными версиями InterBase, а также платформами и ОС. Миграция заключается в том, что в системе-источнике (где система – это уникальное сочетание версии InterBase-сервера, ОС и аппаратной платформы, например InterBase 5.6 +Windows NT+Intel) данные из базы данных "складываются" в некоторый универсальный формат, из которого затем они разворачиваются в базу данных в системе–приемнике с учетом ее особенностей. Универсальный формат должен "пониматься" всеми модификациями InterBase, конечно, с ограничениями на переносимость между старыми и новыми версиями.

Этим универсальным промежуточным хранилищем для данных является резервная копия базы данных (backup), созданная в виде transportable backup (подробнее о процессе резервного копирования и восстановления см. в этой части главу "Резервное копирование и восстановление из резервной копии").

В самом простом случае миграция заключается в том, чтобы создать backup на системе-источнике (где система – это сочетание InterBase+ОС+платформа) и восстановить эту резервную копию на системе-приемнике. Такой способ применим в случае перехода с предыдущей версии InterBase на новую (например, с 5.x на 6.x). Примером более сложного случая миграции является смена диалекта базы данных с 1-го на 3-й, где необходимо либо применять инструмент модификации базы данных gfix, либо полностью пересоздавать базу данных.

Миграция между различными версиями InterBase

Карта миграции

В этом разделе мы рассмотрим, как осуществить процесс миграции с одной версии InterBase на другую. В таблице 4.6 представлены карта возможных переходов с одной версии InterBase на другую.

Под прямой миграцией понимается процесс, включающий backup на системе-источнике и восстановление на системе-приемнике.

> Прямая миграция – это процесс перехода между версиями, который включает следующие этапы: backup (с контрольным restore) - > Установка новой версии IB -> Перенос пользователей -> restore.

Таблица 4.6. Карта миграции

НА ВЕРСИЮ	С ВЕРСИИ			
	InterBase 4.x	InterBase 5.x	InterBase 6.x и клоны (диалект база данных 1)	InterBase 6.x и клоны (диалект база данных 3)
InterBase 4.x	Да	Особый процесс	Особый процесс	Нет
InterBase 5.x	"	Да	Особый процесс	"
InterBase 6.x и клоны (диалект база данных 1)	"	"	Да	"
InterBase 6.x и клоны (диалект база данных 3)	Нет	Нет	"	Да

В случае, если перенос между версиями возможен в виде прямой миграции, в ячейке, соответствующей переходу (пересечению графы и строки) с версии-источника на версию-приемник, ставится "Да". Версия-источник выбирается в заголовке таблицы, версия-приемник – в боковике таблицы. В трех ячейках, соответствующих переходу со старшей версии на младшую, стоит "Особый процесс". Это означает, что процесс миграции возможен с применением особого приема, который мы рассмотрим ниже. Обратите внимание, что переход на версию 6.x с диалектом 3 невозможен с помощью прямой миграции. Для установки диалекта 3 применяется инструмент командной строки gfix (подробнее см. ниже, в разделе "Перевод базы данных InterBase 6.x на 3-й диалект").

Прямая миграция

Итак, чтобы осуществить миграцию между теми версиями InterBase, на пересечении которых в таблице "Карта прямой миграции" стоит "Да", необходимо выполнить следующие действия.

Сделать резервную копию базы данных на системе-источнике. Для этого следует выполнить команду

```
gbak –b –user SYSDBA –password <пароль> <путь к базе данных>
<путь к backup>
```

При этом создастся резервная копия вашей базы данных (подробнее о резервном копировании см. в этой части главу "Резервное копирование базы данных и восстановление из резервной копии"). После этого необходимо проверить правильность этой резервной копии – попытаться восстановить ее на той же самой системе-источнике (но ни в коем случае не поверх базы данных источника!). Для этого выполняем команду:

```
gbak –c –user SYSDBA –password <пароль>  <путь к backup> <путь2
к базе данных>
```

Если контрольное восстановление прошло успешно, то сохраняем резервную копию базы данных в надежном месте (переустанавливаемый сервер таким местом обычно не является, скопируйте backup базы данных на другой компьютер или на какое-нибудь устройство резервного копирования) и приступаем к следующему этапу миграции, если нет, то смотрим главу "Починка базы данных" (см. ниже), чиним свою базу данных и вновь приступаем к процессу миграции с самого начала.

Сохранение информации о пользователях при миграции

Далее, после получения корректной резервной копии вашей рабочей базы данных, необходимо установить новую версию InterBase. Процесс установки описан в главе "Установка InterBase" (ч. 1), и в этой главе мы останавливаться на этом не будем. Но при установке новой версии в рамках миграции вне зависимости от того, куда устанавливается новая версия сервера InterBase – на новый компьютер или на тот же, где стояла предыдущая версия, необходимо позаботиться о перенесении пользователей InterBase на новый сервер (конечно, если в вашей системе применяются еще какие-то вами созданные пользователи помимо устанавливаемого по умолчанию пользователя SYSDBA). Пользователи хранятся отдельно от вашей базы данных – для их хранения существует база данных ISC4.gdb, которая находится в том же каталоге, где установлен InterBase.

> Помните, что, хотя пользователи хранятся в отдельной базе данных ISC4.gdb, все разрешения и права для них хранятся в той же базе, где и объекты, на которые выдавались разрешения (т. е. в самой рабочей базе данных). Все эти права сохраняются при переходе на новую версию InterBase (т. е. они не исчезают при backup/restore). Подробнее о пользователях и правах см. ниже главу "Безопасность в InterBase: пользователи, роли и права".

При переустановке поверх старой версии установщик InterBase очень мудро не затирает существующие ISC4.gdb (а также ISC4.gbk), чтобы ненароком не стереть информацию о пользователях. Однако, несмотря на такую предусмотрительность, могут возникнуть проблемы, связанные с тем, что новый сервер может не суметь прочитать оставшуюся в наследство ISC4.gdb из-за различия в структурах баз данных новой и старой версии InterBase.

Чтобы избежать потерь информации и других проблем с базой данных пользователей ISC4.gdb при переустановке InterBase, надо сделать следующее:

- до установки новой версии сделать backup ISC4.gdb с использованием старой версии InterBase;
- в случае установки новой версии поверх старой переместить ISC4.gdb из установочного каталога InterBase, чтобы она не помешала установщику InterBase записать туда свою ISC4.gdb, которая создается по умолчанию при новой установке;
- после установки новой версии InterBase надо восстановить базу данных пользователей из созданной резервной копии старой ISC4.gdb и заменить ею ту, которая была создана по умолчанию при установке новой версии.

Рассмотрим теперь этот процесс подробнее. Для резервного копирования ISC4.gdb можно воспользоваться командой вроде этой:

```
gbak –b –user SYSDBA –password <пароль> C:\IBServer\isc4.gdb
C:\isc4.gbk
```

Для восстановления следует воспользоваться тем, что при установке новой версии всегда создается пользователь SYSDBA (с паролем по умолчанию masterkey) и мы можем восстановить backup старой ISC4.gdb:

```
gbak –c –user SYSDBA –password masterkey  C:\isc4.gbk
C:\isc4.gdb
```

а затем заменить восстановленной копией ту ISC4.gdb, которая сформировалась по умолчанию в результате установки:

```
copy C:\isc4.gdb <путь к каталогу с новой версией IB>\isc4.gdb /y
```

Перед процедурой копирования восстановленной ISC4.gdb на положенное место желательно остановить сервер InterBase.

Восстановление из резервной копии на системе-приемнике

Итак, мы установили новую версию InterBase и перенесли на нее информацию о наших пользователях (т. е. восстановили базу данных ISC4.gdb). Теперь мы готовы восстановить резервную копию рабочей базы данных (созданную на старой системе) в новой версии InterBase. Для этого можно воспользоваться командой, похожей на эту:

```
gbak –c –user SYSDBA –password <пароль>  <путь к backup>
<путь2 к базе данных>
```

Как видите, эта команда аналогична той, которую мы применяли для контрольного восстановления резервной копии. База данных восстановится на новой системе, причем если вы сменили версию сервера (например, с 5.x на 6.x), то во время восстановления базы данных будет создана уже с новой версией ODS, т. е. прямая миграция "назад", на предыдущую версию, будет невозможна.

При восстановлении вы можете изменить владельца базы данных, т. е. можно восстановить база данных от имени какого-либо другого пользователя, а не от SYSDBA. Это бывает полезным в целях повышения безопасности работы с базой данных.

Аналогичный процесс миграции применяется и в случае смены аппаратной платформы и/или ОС – будь то переход на другую платформу (Intel->Sparc) или просто замена оборудования или ОС.

Прямая миграция по вышеописанному алгоритму – наиболее простой и частый вид миграции баз данных InterBase. Однако, как вы можете видеть, в таблице 4.6 существуют также миграции, обозначенные как "Особый процесс". Сейчас мы подробно рассмотрим его.

Особый процесс, или обратная миграция

Особый процесс выражать такой переход между версиями InterBase, когда обычным методом, через backup/restore, базу данных не удастся "понизить" до младшей версии: gbak от младшей версии откажется работать с базами данных, созданными с использованием старшей версии InterBase.

> В целом обратная миграция является недокументированным действием, и потому особых гарантий целостности данных при таком переходе дать нельзя, однако известно достаточно много успешных примеров подобного переноса.

Чтобы осуществить обратную миграцию базы данных, необходимо задействовать два компьютера с установленными на них серверами InterBase, один из которых имеет новую версию (источник), а другой – предыдущую (приемник). Последовательность действий такая:

На компьютере-приемнике запускаем инструмент командной строки gbak и даем ему указание создать backup базы данных, находящейся на компьютере-источнике. Например, если компьютер-источник называется source_nt, то команда будет выглядеть примерно так:

```
gbak –b –user SYSDBA –password <пароль> source_nt:<путь_к_базе
данных_источнику> <Путь к backup-приемнику>
```

При этом gbak подключится к серверу-источнику как клиент (здесь используется возможность обратной совместимости, когда клиент младшей версии может подсоединиться к серверу, имеющему старшую) и произведет чтение всех данных из базы данных-источника, пользуясь возможностями старшей версии сервера. Но backup этой базы данных будет создан с использованием старой версии InterBase-приемника, т. е. полученную резервную копию в дальнейшем можно будет восстановить в полноценную базу данных, соответствующую младшей версии. Естественно, при таком подходе могут быть "подводные камни" – в тех случаях, когда в базе данных используются те свойства новой версии, которые не поддерживаются в старой. При этом возможно несколько исходов процесса обратной миграции: gbak либо проигнорирует новые возможности и попытается закончить резервное копирование без них, либо попытается проинтерпретировать их в стиле своей версии и получить какие-то правдоподобные значения. Конечно, наличие новых свойств в базе данных, которую переводим на младшую версию InterBase, таит в себе ту опасность, что, произведя обратную миграцию, мы окажемся у "разбитого корыта" – с базой данных, наполненной некорректными значениями или вообще нечитабельной. К счастью, gbak достаточно жестко относится к неоднозначностям в процессе backup: практически всегда, когда он наталкивается на неизвестную ему особенность, выдается ошибка. Это предохраняет от возможных скрытых ошибок в интерпретации данных. Например, при наличие в базы данных от InterBase 6.x 64-разрядных генераторов при попытке перевода этой базы на InterBase 5.x возникнет ошибка, сигнализирующая о том, что в 5.x нет подобных генераторов. Их придется удалить перед обратной миграцией, а после восстановления базы данных вновь создать.

Вот вкратце мы и рассмотрели практически все возможные варианты миграции, приведенные в таблице 4.6. Остался открытым вопрос переводе базы данных InterBase 6.x с диалекта 1 на диалект 3. Мы вернемся к нему чуть позже, а пока рассмотрим поведение и вопросы совместимости клиентов и серверов InterBase различных версий.

Совместимость клиентов и серверов различных версий

Как известно, клиент-серверное приложение, использующее СУБД InterBase, обычно состоит из двух основных частей – клиентской и серверной. Клиентская часть, обычно состоящая из исполняемого модуля приложения базы данных (как правило, exe-файла) и динамических библиотек, исполняется на клиентском компьютере. Серверная часть – собственно базы данных и серверные модули InterBase, обслуживающие клиентские запросы, – исполняется на компьютере-сервере (подробнее о составе модулей, образующих клиент и сервер СУБД InterBase, см. ниже главу "Состав модулей InterBase").

То, что в результате переустановки InterBase и миграции базы данных вы смените версию сервера InterBase на компьютере-сервере, не означает, автоматической замены клиентов на всех клиентских компьютерах. Обычно необходимо вручную заменить клиентскую часть InterBase на этих компьютерах. То есть обычно клиенты InterBase должны иметь ту же версию, что и сервер. Однако если вы не замените клиентские части, то можете обнаружить, что клиенты от младшей версии InterBase работают со старшей версией InterBase-сервера. Такая совместимость – вполне нормальная, документированная особенность, которая используется для облегчения процесса миграции. Возможность использования клиентов и серверов InterBase различных версий представлена в таблице 4.7.

Таблица 4.7. Совместимость клиентов и серверов различных версий InterBase

Сервер от версии /диалект БД	Клиент версии		
	InterBase 4.x	InterBase 5.x	InterBase 6.x и клоны
InterBase 4.x	Полная	Полная	Полная
InterBase 5.x	С ограничениями	"	"
InterBase 6.x и его клоны /диалект 1	Неустойчивая	С особыми свойствами	"
InterBase 6.x и его клоны /диалект 3	"	"	"

В таблице 4.7 на пересечении версии клиента и версии сервера (для 6.x – с учетом диалекта базы данных) стоит описание их совместимости.

Как видно, совместимость клиентов и серверов InterBase имеет 4 основных разновидности – полную, с ограничениями, с особыми свойствами и неустойчивую.

Полная совместимость возможна в ситуации, когда клиент и сервер имеют одинаковые версии или когда версия клиента старше версии сервера. Полная совместимость означает, что клиент может реализовать всю функциональность, предлагаемую сервером.

Совместимость с ограничениями означает, что, хотя сервер и предлагает большую функциональность, чем может поддержать клиент, их взаимодействие все же вероятно – в рамках тех возможностей, которые поддерживает клиент.

Неустойчивая совместимость означает, что взаимодействие слишком "старого" клиента и сервера возможно, но в ряде случаев такое взаимодействие может привести к ошибкам сервера или клиента, а также к порче данных. Такое сочетание клиента и сервера использовать не рекомендуется.

Совместимость с особыми свойствами возникает, когда клиенты версии 5.x связываются с базами данных, работающими под управлением InterBase-сервера версии 6.x. Причем тот факт, имеет ли база данных диалект 1 или диалект 3, имеет большое значение.

Если клиент от версии 5.x соединяется с базой данных 6.x, имеющей диалект 1, то он получает возможность работать с этой базой так, как будто она находится под управлением InterBase-сервера 5.x. Это означает (помимо невозможности использовать новые свойства InterBase 6.x), что, если в переведенной из-под 5.x базе данных есть объекты, название которых совпадает с каким-либо новым ключевым словом InterBase 6.x, клиенты 5.x по-прежнему будут иметь возможность работать с этой базой данных, используя ключевые слова в качестве идентификаторов. Но только для доступа к уже существующим объектам. Создание новых объектов, использующих в качестве идентификаторов ключевые слова, невозможно ни в клиентах 5.x, ни в клиентах 6.x.

Вот список новых ключевых слов, появившихся в InterBase 6.x:

COLUMN, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, DAY, EXTRACT, HOUR, MINUTE, MONTH, SECOND, TIME, TIMESTAMP, TYPE, WEEKDAY, YEAR, YEARDAY.

Клиенты InterBase 5.x, подсоединяющиеся к базе данных 6.x, которая имеет диалект 3, имеют следующие ограничения:

- отсутствие доступа к полям, имеющие новые типы данных, определенные в 3-м диалекте InterBase 6.x;
- отсутствие доступа к идентификаторам, заключенным в кавычки;
- поля, имеющие тип DATE, рассматриваются клиентами 5.x как тип TIMESTAMP, т. к. в 4.x и 5.x тип "дата+время" назывался DATE.

Клиенты, применяющие для доступа к базе данных BDE версии ниже 5.3, не могут использовать объекты, имеющие новые типы данных, появившихся в 3-м диалекте InterBase 6.x.

Подводя итог разделу о совместимости клиентов и серверов, необходимо добавить, что, несмотря на возможность использовать "старых" клиентов, лучше всего использовать клиентов той же версии, которую имеют и серверы. Причем желательно, чтобы это соответствие было точным – вплоть до номеров билдов. То есть если вы используете в качестве сервера какой-нибудь клон InterBase, например Firebird 1.0, то желательно использовать и клиента именно от этой версии, а не от InterBase 6.0.1, например.

Перевод базы данных InterBase 6.x на 3-й диалект

Итак, мы подходим к рассмотрению вопроса о переводе баз данных InterBase 6 1-го диалекта на диалект 3, использующий все возможности версии 6.x. Начнем рассматривать этот перевод с предположения, что в качестве исходного материала мы имеем базу данных InterBase 6.x диалекта 1 и клиентские приложения, использующие синтаксис 1-го диалекта.

Сейчас речь идет именно о переводе базаы данных в 3-й диалект. Дело в том, что клиентские приложения при подключении к базе данных также указывают диалект. После рассмотрения перевода базы данных мы рассмотрим и те шаги, которые нужны для миграции приложения на 3-й диалект.

Итак, что же мешает нам перевести нашу базу данных в 3-й диалект? Существует 4 основных препятствия, которые надо преодолеть, чтобы успешно перейти на 3-й диалект, – это двойные кавычки, ключевые слова, новые типы данных DATE и большие целые числа.

Двойные кавычки

Если в версиях InterBase 4.x и 5.x и диалекте 1 версии 6.x строковые константы позволялось описывать как с помощью как одинарных, так и двойных кавычек, то в 3-м диалекте двойные кавычки применяются только для обозначения идентификаторов, а одинарные – для строковых констант. Чтобы базу можно было перевести из 1-го диалекта в 3-й, необходимо заменить все двойные кавычки, ограничивающие строковые константы, на одинарные. Двойные кавычки могут находиться в триггерах, хранимых процедурах, представлениях, доменах, ограничениях и в значениях столбцов по умолчанию.

Ключевые слова

Новые ключевые слова могли быть использованы в базе данных, созданной в InterBase 4.x или 5.x, в качестве идентификаторов каких-либо объектов. Для перехода к диалекту 3 необходимо переименовать эти объекты, например YEAR -> YEAR1 или YEAR->"year" (в 3-м диалекте ключевые слова могут быть идентификаторами, если они заключены в двойные кавычки). При смене диалекта базы данных лучше не использовать кавычки для разрешения конфликтов с ключевыми словами: разработчику надо привыкнуть к такому стилю работы и запомнить, что "Year", "YEAR" и "year" – разные идентификаторы.

Типы данных для работы с датой и временем

В версиях 4.x и 5.x и 1-м диалекте 6.x присутствует только один тип данных – DATE, который позволяет хранить информацию о дате и времени суток. В 3-м диалекте существует 3 типа для работы с датой и временем – это тип TIMESTAMP, хранящий информацию о дате и времени (т. е. он аналогичен типу DATE в 1-м диалекте и в 5.x), тип DATE, хранящий информацию только о дате, и тип TIME, хранящий информацию только о времени. Довольно запутанно,

не так ли? Функции "старого доброго" DATE теперь берет на себя TIMESTAMP, а DATE, изменив свое назначение, означает теперь только дату.

Как же происходит замена типов данных? Объявления столбцов типа DATE автоматически меняют свой тип на TIMESTAMP уже при переходе на диалект 1 InterBase 6x, а вот с переменными типа DATE, использующимися (может быть) в триггерах и процедурах, ситуация сложнее. Они не меняют свой тип автоматически, и необходимо будет сменить их тип ("старый" DATE) на TIMESTAMP перед сменой диалекта 1 на 3.

Большие целые типы

В 3-м диалекте целые числа, имеющие тип NUMERIC или DECIMAL и разрядность больше девяти, хранятся в виде INT64, а менее девяти – в виде DOUBLE PRECISION. В 1-м диалекте и старых версиях все большие целые числа хранятся как DOUBLE PRECISION.

> Обратите внимание, что INT64 – это обозначение механизма хранения больших целых чисел, а не какой-то конкретный тип.

При переходе с 1-го диалекта на 3-й НИЧЕГО не изменится в хранении больших целых чисел, созданных ДО перехода на 3-й диалект, – они по-прежнему будут иметь тип DOUBLE PRECISION. Чтобы воспользоваться преимуществами хранения данных в INT64 (подробности см. в главе "Типы данных" (ч. 1)), можно перенести данные в столбец с типом INT64. Для переноса нужно создать новый столбец нужной разрядности (например, NUMERIC(15,2)), который будет хранить свои значения в виде INT64, и перенести туда значения из старого столбца, затем старый столбец-источник удалить, а новый (со значениями в INT64) переименовать как старый. Переименование легко осуществить, воспользовавшись командой ALTER COLUMN, которая может сменить имя, тип и позицию столбца.

Вот такие препятствия ждут нас на пути к 3-му диалекту. Теперь от обзора перейдем к пошаговому алгоритму перевода базы данных от 1-го диалекта к 3-му. Мы будем рассматривать вариант, названный в [3, глава "Migration Guide"] способом "In-place migration", – когда база данных переводится на 3-й диалект без полного пересоздания базы данных и перекачки всех данных из старой базы данных в новую.

Пошаговые инструкции для перехода на 3-й диалект

Исходные данные: предположим, что мы переводим базу данных, состоящую из одного файла, – base2migrate.gdb. База данных – источник имеет 1 диалект. Резервная копия базы данных-источника сделана.

Итак, начнем.

1. Необходимо получить метаданные, описывающие базу данных-источник в 1-м диалекте. Для этого выполняем команду вида isql base2migrate.gdb –x –user SYSDBA -password masterkey -o baseSource.ddl, которая извлечет метаданные и поместит их в файл baseSource.ddl. Эта команда выполнится успешно, если

база данных-источник находится в том же каталоге, в котором находится и isql, в другом случае придется указать полный путь к базе данных.

2. Необходимо создать пустой файл с именем вроде makeIt.sql, в который будут заноситься команды изменения метаданных базы данных, которые получим на основе анализа файла baseSource.ddl на предмет несоответствия содержащихся там метаданных требованиям диалекта 3. Команды изменения из файла makeIt.sql подготовят базу данных-источник к переходу. Дальнейшие шаги до перехода будут посвящены наполнению файла.
3. Отыскиваем в файле baseSource.ddl все двойные кавычки и заменяем их на одинарные. Затем копируем все измененные выражения в файл makeIt.sql. Например, если строка, ограниченная двойными кавычками, находится внутри триггера, то надо скопировать весь триггер с измененной строкой (и не забудьте предложение set term перед триггером!). Заметьте, что простым поиском/заменой кавычек здесь не обойтись, ведь двойные и одинарные кавычки могут быть и в середине, и в конце, и в начале строковых констант. Для замены следует воспользоваться следующими правилами, составленными на основе таблицы 2.1 в InterBase 6 Mignation Guide [3]:

Таблица 4.8. Правила перевода строк с двойными кавычками

Двойные кавычки внутри строки	
Строка без кавычек как она есть	In "peg" mode
Строка, заключенная в двойные кавычки (допускается правилами IB5.x и 1-го диалекта)	"In ""peg"" mode"
Строка, заключенная в одинарные кавычки согласно требованиям правил диалекта 3	'In "peg" mode'
Одинарные кавычки внутри строки	
Строка без кавычек	O'Reilly
Строка, заключенная в двойные кавычки (допускается правилами IB5.x и 1-го диалекта)	"O'Reilly"
Строка, заключенная в одинарные кавычки, согласно требованиям правил диалекта 3	'O''Reilly'

4. После разрешения вопросов с двойными кавычками необходимо разобраться в типах для работы с датой и временем. Во время перехода на диалект 1 все "старые" столбцы типа DATE заменились на TIMESTAMP. Однако переменные типа DATE, которые могли быть в триггерах и процедурах, автоматически не заменились на TIMESTAMP. Поэтому надо произвести в файле baseSource.ddl поиск всех вхождений переменных типа DATE и сменить их тип на TIMESTAMP. Все предложения (триггеры, представления, хранимые процедуры и т. д.), затронутые изменениями, следует перенести в файл

makeIt.sql. А затем надо повторить такой же поиск и произвести замену DATE на TIMESTAMP в файле makeIt.sql, ведь в этот файл уже попали несколько предложений, модифицированных для того, чтобы соответствовать правилам 3-го диалекта для работы с двойными кавычками, и эти предложения тоже надо очистить от нежелательных переменных типа DATE.

5. Теперь следует заняться ключевыми словами, которые могли быть использованы в базе данных версий 5.x. Необходимо в файле makeIt.sql найти все предложения, содержащие ключевые слова и изменить их на неключевые обозначения. Возможно, что изменение наименований столбцов повлечет за собой необходимость удалить и пересоздать зависимые от них объекты. Для получения списка зависящих от каждого меняющегося столбца объектов можно воспользоваться инструментами, указанными в приложении "Инструменты администратора и разработчика InterBase". Если существуют зависимые объекты, то их придется сначала удалить, а потом создать вновь, но уже с правильными ссылками на измененные объекты. То есть если у вас было поле, названное YEAR, и вы сменили его имя на YEAR1, то во всех процедурах, представлениях, триггерах необходимо заменить YEAR на YEAR1. Для этого придется удалить все эти объекты из базы данных, затем исправить имя столбца и после этого пересоздать все зависимые объекты.

   Вообще говоря, избавление от ключевых слов является наиболее трудоемкой процедурой из-за необходимости отследить и пересоздать зависимые объекты. Лучше всего работу по удалению ключевых слов из базы данных выполнить отдельно, а не в рамках того набора изменений, который формируется в файле makeIt.sql.

   После разрешения вопроса с ключевыми словами в файле makeIt.sql ту же самую процедуру необходимо проделать в файле baseSource.ddl и, как и раньше, скопировать все измененные предложения в файл makeIt.sql.

6. Теперь необходимо превратить предложения, содержащиеся в файле makeIt.sql, в команды DDL, которые приведут нашу базу данных в соответствие с правилами 3-го диалекта.
   Для этого в файле makeIt.sql надо заменить словосочетание CREATE TRIGGER на ALTER TRIIGER, CREATE DOMAIN – на ALTER DOMAIN. Затем перед каждой командой CREATE VIEW вставить DROP VIEW для создаваемого представления – сочетание DROP/CREATE используется из-за отсутствия для представлений команды ALTER. Проверьте, чтобы среди скопированных предложений не оказалось команд CREATE PROCEDURE, а были только ALTER PROCEDURE. Затем если в makeIt.sql есть предложения ALTER TABLE, которые изменяют ограничения таблиц (CHECK), то модифицируйте предложения ALTER TABLE так, чтобы они сначала удаляли эти константы, а затем вновь создавали, но уже с одинарными кавычками. Вы можете заметить, что измененные из-за двойных кавычек предложения могут дублироваться в файле makeIt.sql предложениями, имеющими переменные типа DATE, но это неопасно: повторное выполнение предложений по модификации не приведет к порче метаданных, хотя, возможно, приведет к возникновению ошибок вида "attempt to store duplicate value", которые сигнализируют о попытке повторного создания объекта.

7. В начало файла makeIt.sql поместите команду CONNECT, которая обеспечит подключение к модифицируемой базе данных – что-то вроде:
```
CONNECT 'C:\Database\base2migr.gdb'  USER 'SYSDBA' PASSWORD
'masterkey';
```
8. Запустите скрипт из файла makeIt.sql с помощью isql или любого другого инструмента администрирования базы данных. Вполне возможно, что первый запуск этого файла приведет к появлению массы ошибок. Внимательно проанализируйте эти ошибки, внесите соответствующие изменения в makeIt.sql и вновь запустите его на выполнение, только уже не на "использованной" базе данных, – извлеките из backup новую, нетронутую базу данных и проверяйте скрипт на ней.
9. Когда добьетесь удовлетворительного результата со скриптом в makeIt.sql, необходимо явно установить 3-й диалект базы данных следующей командой:
```
gfix -sql_dialect 3 base2migr.gdb -user sysdba -password mas-
terkey
```

Теперь вы получили базу данных 3-го диалекта. Конечно, ряд моментов остался нерассмотренным, поэтому в случае возникновения необходимости получить исчерпывающую информацию по переходу на 3-й диалект следует обратиться к [3, глава "Migration Guide"].

Клиенты 3-го диалекта

Переведя базу данных на 3-й диалект, необходимо также перевести на него и клиентские приложения. Обычно в параметрах подключения к базе данных необходимо указывать диалект, с помощью которого будет производиться работа с базой данных. Некоторые библиотеки доступа к InterBase, например IBProvider, автоматически определяют диалект базы данных и самостоятельно выставляют нужные параметры подключения.

Установив параметры подключения, необходимо привести в соответствие с правилами 3-го диалекта текст SQL-запросов, содержащихся в клиентском приложении. Необходимо изменить все ключевые слова в текстах запросах согласно тем изменениям в базе, которые были произведены во время миграции.

Также необходимо позаботиться о хранении больших целых чисел (хранящихся с использованием механизма INT64), если таковые присутствуют в базе данных и используются в качестве параметров или результатов запросов.

Заключение

Главное, что следует вынести из этой главы, – это то, что миграция процесс сложный и нелинейный, поэтому нельзя браться за нее без тщательного продумывания и, конечно же, без резервного копирования.

Починка базы данных

Обзор основных причин повреждения базы данных

К сожалению, всегда существует ненулевая вероятность, что любое информационное хранилище будет повреждено и часть информации из него потеряна. Базы данных не исключение из этого правила. В этой главе мы рассмотрим основные причины, которые чаще всего приводят к повреждениям базы данных InterBase, рассмотрим несколько способов восстановления баз данных и извлечения из них информации. Также ознакомимся с рекомендациями и профилактическими действиями, которые позволят свести к минимуму риск потери информации из базы данных.

Прежде всего, раз мы говорим о починке базы данных, необходимо определиться с понятием "поломка базы данных". Обычно базу данных называют поврежденной, если при попытке извлечь или модифицировать содержащуюся в ней информацию возникают ошибки и/или извлекаемая информация оказывается утерянной, неполной или вовсе неправильной. Порой повреждения базы данных скрыты и обнаруживаются только при проверке специальными средствами, но бывают и явные поломки базы данных, когда к базе невозможно подсоединиться, когда отлаженные программы-клиенты выдают странные ошибки (в то время как никаких манипуляций над базой данных не производилось) или когда невозможно восстановить базу данных из резервной копии.

Основными причинами повреждения баз данных являются:

1. Аварийное завершение работы серверного компьютера, особенно отключение электропитания. Для российской информационной отрасли это настоящий бич, поэтому мы надеемся, что не нужно лишний раз напоминать о необходимости иметь на сервере источник бесперебойного питания.
2. Дефекты и неисправности серверного компьютера, особенно дисков, дисковых контроллеров, оперативной памяти компьютера и кеш-памяти RAID-контроллеров.
3. Некорректное соединение с многопользовательской базой данных одного или более пользователей. При соединении по протоколу TCP/IP путь к базе данных должен указываться servername:drive:/path/databasename (для серверов на платформе Unix servername:/path/databasename), по протоколу NetBEUI \\servername\drive\path\databasename. Даже при соединении с базой с того же компьютера, на котором находится база и работает сервер, следует пользоваться точно такой же строкой, заменяя servername на localhost. Нельзя использовать mapped drive в строке соединения. При нарушении любого из этих правил сервер считает, что он работает с разными базами, и повреждение базы данных гарантировано.
4. Файловое копирование или другой файловый доступ к базе данных при запущенном сервере. Выполнение команды shutdown или отключение пользователей обычным порядком не является гарантией того, что сервер ничего

не делает с базой; если sweep interval не установлен в 0, может выполняться sweep. Кроме того, после отключения последнего пользователя сервер выполняет уборку "мусора". Обычно на это уходит 1–2 мин, но, если перед этим выполнялось много операций delete или update, процесс может затянуться.

5. Использование нестабильных серверов InterBase 5.1–5.5. Компания Borland официально признала наличие в этих серверах серьёзных ошибок и выкладывала на своём сайте для бесплатного скачивания покупателями серверов 5.1–5.5 стабильный upgrade 5.6, убранный только после выпуска сертифицированного InterBase 6.
6. Превышение ограничения на размер файла базы. Для большинства существующих на момент написания этих строк серверов Unix-платформы это 2 Гбайт, для Windows NT/2000 – 4 Гбайт, но рекомендовано ориентироваться также на 2 Гбайт. При приближении размера базы к граничному значению должен быть создан дополнительный файл.
7. Исчерпывание свободного дискового пространства во время работы с базой.
8. Для Borland InterBase-серверов версий меньше 6.0.1.6 превышение ограничения на количество генераторов, по сообщению Borland InterBase R&D, определяемое следующим образом (см. таблицу 4.9).

Таблица 4.9. Критическое количество генераторов в InterBase ранних версий

Версия	Размер страницы, байт			
	1024	2 K	4 K	8 K
Pre-V6	248	504	1016	2040
V6	124	257	508	1020

9. Для всех серверов Borland InterBase превышение допустимого количества транзакций без выполнения backup/restore. Узнать количество транзакций, произошедших в базе данных с момента последнего создания (или restore), можно с помощью вызова утилиты gstat с ключом –h, параметр NEXT TRANSACTION ID будет искомым числом транзакций. По сообщению Ann W. Harrison, критическое количество транзакций зависит от размера страницы и имеет следующие значения (см. таблицу 4.10).

Таблица 4.10. Критическое количество транзакций в серверах Borland InterBase

Размер страницы базы данных, байт	Критическое число транзакций
1024	131 596 287
2048	265 814 016
4096	534 249 472
8192	1 071 120 384

Перечисленные выше ограничения серверов Borland InterBase не распространяются на сервера Firebird за исключением самых ранних версий 0.x, существование которых стало уже историей. Если вы используете окончательную версии

(релиз) Firebird 1.0 или InterBase 6.5, то вам не следует беспокоиться о пп. 5, 6, 8 и 9, а надо сосредоточить свои усилия на остальных причинах. Сейчас мы подробно рассмотрим наиболее частые из них.

Отключение питания

При отключении питания на компьютере-сервере все процессы обработки данных прерываются в самых неожиданных и (согласно закону Мерфи) опасных местах. В результате информация в базе данных может исказиться или вовсе пропасть. Самый простой случай, когда в результате отключения питания все неподтвержденные данные из пользовательских программ-клиентов пропали. После восстановления питания сервер просматривает данные, видит незавершенные транзакции, не привязанные ни к одному из "живых" клиентов, и откатывает все изменения, проведенные в рамках этих "погибших" транзакций. Собственно, такое поведение является нормальным и изначально предполагаемым разработчиками InterBase. Однако отключение питания не всегда сопровождается лишь такими незначительными потерями. Если сервер в момент отключения питания производил расширение базы данных, то велик риск получить "потерянные" страницы в файле базы данных (orphan pages), т. е. такие страницы, которые физически распределены и зарегистрированы на страницах учета страниц (PIP), но запись данных на которые невозможна. Подробнее о потерянных страницах см. ниже главу "Структура базы данных InterBase". Бороться с потерянными страницам в файле базы данных умеет только инструмент починки и модификации gfix, который мы подробнее рассмотрим ниже. Собственно, потерянные страницы приводят только к излишнему расходу дискового пространства и как таковые не служат причиной потери или порчи данных. Но потеря питания приводит и к более серьезным повреждениям.

Например, после отключения питания и повторного включения может оказаться, что пропало большое количество данных, в том числе и подтвержденных (после добавления или модификации которых была выполнена команда подтвердить транзакцию, – т. е. commit). Это происходит из-за того, что подтвержденные данные записываются не напрямую в файл базы данных на диске, а используют для этой цели файловый кеш ОС. То есть серверный процесс передал ОС команду на запись данных на диск, ОС "успокоила" сервер, что данные сохранены на диске, а на самом деле данные находятся в файловом кеше. ОС не торопится сбрасывать эти данные на диск, так как оценивает, что оперативной памяти еще много, и откладывает медленные операции записи на диск до тех пор, пока не закончится свободная оперативная память.

Forced writes – палка о двух концах

Чтобы влиять на эту ситуацию, в InterBase 6 предусмотрена настройка режима записи данных на диск. Эта настройка называется *forced writes* (FW) и имеет два значения – ON (synchronous) и OFF (asynchronous). Значения forced writes определяют, каким образом InterBase взаимодействует с диском. Если установлено значение forced writes on, то включается режим синхронной записи на диск, когда подтвержденные данные записываются на диск сразу после команды

commit, причем сервер ждет завершения записи и лишь потом продолжает работу. В случае режима forced writes off InterBase не торопится записывать данные на диск после команды подтверждения транзакции (commit), а делегирует эту задачу параллельному потоку, в то время как основной поток продолжает обработку данных, не дожидаясь конца записи на диск.

Режим синхронной записи на диск (FW ON) является более безопасным и приводит к минимизации возможной потери данных, однако следствием его применения является некоторая потеря производительности. Режим асинхронной записи (FW OFF) увеличивает производительность, однако значительно возрастает риск потери большого количества данных.

Для получения максимально возможной производительности обычно устанавливают режим FW OFF, в результате чего при сбое питания теряется гораздо большее количество данных, чем при синхронном режиме записи на диск. При установке режима записи следует хорошенько взвесить, насколько увеличение производительности важнее возможности потерять несколько часов работы при неожиданном сбое питания.

Часто сами пользователи грешат неджентльменским отношением к InterBase. В небольших организациях, где экономят на любых мелочах, зачастую на компьютер-сервер, на котором установлен сервер СУБД, также ставят другие серверные (и не только серверные) программы. И в случае их "зависания", недолго думая, нажимают на Reset, что повторяется несколько раз на дню. Хотя InterBase является необычайно устойчивым к таким действиям по сравнению с другими СУБД и позволяет начать работу с базой данных сразу после аварийной перезагрузки, однако такое обращение не проходит бесследно. В результате аварийных перезагрузок накапливаются потерянные страницы, теряются связи между страницами данных. Это может продолжаться довольно долго, однако развязка рано или поздно наступит. Когда "битые страницы" появятся среди страниц учета страниц (PIP) или затронут страницы генераторов или, еще хуже, испортится заголовочная страница базы данных, то база данных может просто больше не открыться и превратиться в большой кусок разрозненных данных, из которого нельзя извлечь ни байта полезной информации.

Повреждения жесткого диска

К такому же печальному результату могут привести повреждения жесткого диска (появление bad sectors) и нехватка дискового пространства в момент расширения базы данных. В последнем случае может произойти очень неприятная вещь: InterBase укажет на заголовочной странице, что файл базы данных теперь состоит из N страниц, в то время как реальное расширение файла до указанного размера аварийно прервется из-за нехватки дискового пространства. При этом скорее всего произойдет аварийное завершение сервера, затем он запустится снова и попытается подключиться к базе данных. Сравнит количество декларированных страниц в заголовке базы данных с реальным размером файла и откажется работать с таким файлом, выдав сообщение об ошибке "Unexpected end of file". Именно такой случай поломки базы данных мы подробно рассмотрим ниже.

Ошибки проектирования базы данных

Необходимо также рассказать о некоторых ошибках, допускаемых разработчиками базы данных, которые могут привести к невозможности восстановления базы данных из резервной копии (файлы *.gbk, создаваемые программой gbak). Прежде всего это небрежное обращение с ограничениями целостности на уровне базы данных. Типичный пример – это ограничения NOT NULL. Предположим, что мы имеем таблицу, которая заполнена некоторым количеством записей. Теперь мы добавим к такой таблице с помощью команды ALTER TABLE еще один столбец, причем укажем, что он не может содержать неопределенных значений NULL, примерно так:

```
ALTER TABLE sometable ADD Field1 INTEGER NOT NULL
```

И в данном случае не возникнет никакой ошибки, как этого можно было бы ожидать. Эта модификация метаданных выполнится, и мы не получим никаких сообщений об ошибках или предупреждений, что создает иллюзию нормальности данной ситуации. Однако если теперь мы произведем резервное копирование базы данных (backup) и попытаемся восстановить (restore) базу данных из этой резервной копии, то на этапе восстановления получим сообщение об ошибке (о том, что в столбец, имеющий ограничение NOT NULL, вставляются NULL) и процесс восстановления прервется. Эта резервная копия невосстановима. Если же восстановление было направлено в файл, имеющий то же имя, что и существующая база данных (т. е. при восстановлении перезаписывался существующий рабочий файл базы данных), то потеряем всю имеющуюся информацию. Это связано с тем, что ограничения NOT NULL реализуются с помощью системных триггеров, которые проверяют лишь вновь поступающие данные, т. е. "срабатывают" при вставке и модификации записей, а существующие данные обходят своим вниманием. При восстановлении же все данные из резервной копии вставляются в пустые, только что созданные таблицы; вот тут-то и выявляются недопустимые NULL в столбце с ограничением NOT NULL.

Некоторые программисты считают такое поведение InterBase ошибкой, но другое поведение просто не позволяет добавить поле с ограничением NOT NULL к таблице с данными. Вариант с требованием обязательного значения по умолчанию и заполнения им в момент создания публично обсуждался архитекторами Firebird, но не был принят из тех соображений, что программист, очевидно, намерен заполнить его в соответствии с каким-то алгоритмом, в общем случае довольно сложным и, возможно, итеративным. При этом не исключено, что он не будет иметь возможности отличить записи, пропущенные предыдущей итерацией, от незаполненных записей.

Похожий дефект данных может возникать в результате сбоев алгоритма сборки "мусора" из-за некорректного задания пути к базе (причина повреждения, указанная в п. 3) при соединении и при файловом доступе к файлам базы данных во время работы с ней сервера 4). При этом в некоторых таблицах могут появиться записи, целиком заполненные NULL. Выявить такие записи довольно сложно, поскольку они не соответствуют ограничениям контроля целостности и уникальности данных, наложенным на таблицы, и оператор Select их просто "не видит", хотя в резервную копию они попадают. В случае невозможности восстановления по этой причине следует обработать исходную базу программой

gfix (см. ниже), найти и удалить такие записи, используя неиндексированные атрибутные поля в качестве условий поиска, после чего повторить попытку снятия резервной копии и восстановления из нее базы.

Подводя итог, можно сказать, что причин возникновения тех или иных поломок базы данных существует большое количество и всегда следует рассчитывать на худшее, а именно, что база данных по той или иной причине повредится. Значит надо быть готовым ее восстановить и спасти ценную информацию. Далее мы рассмотрим профилактические процедуры, гарантирующие сохранность баз данных InterBase, а также способы починки поврежденных баз данных.

Профилактика повреждений баз данных InterBase

Профилактические действия, которые следует производить для предотвращения поломок баз данных, – это постоянное создание резервных копий (подробно о резервном копировании см. главу "Резервное копирование и восстановление из резервной копии"). Это самый надежный рубеж обороны против повреждений базы данных. Только резервное копирование дает 100 %-ную гарантию сохранности данных. Но, как описано выше, в результате резервного копирования может получиться и невосстановимая, т. е. бесполезная, копия, поэтому восстановление базы из копии никогда не должно выполняться путем записи поверх оригинала и резервное копирование должно следовать определенным правилам.

Во-первых, резервное копирование должно быть как можно более частым, во-вторых, оно должно быть последовательным, и, в-третьих, резервные копии нужно проверять на возможность восстановления.

Частое резервное копирование означает, что нужно достаточно часто делать резервную копию, например один раз в сутки. Чем меньше промежуток данных между резервным копированием базы данных, тем меньше данных потеряется в результате сбоя.

Последовательность резервного копирования означает, что резервные копии должны накапливаться и храниться как минимум неделю. Если есть возможность, то нужно записывать резервные копии на специальные устройства типа стримера, если нет – то хотя бы скопировать их на другой компьютер. История резервных копий позволит отследить скрытые повреждения и справиться с ошибкой, которая возникла давно, а проявилась, как всегда, неожиданно.

Необходимо проверять, можно ли восстановить без ошибок полученную резервную копию. Проверить это можно только одним способом – через тестовый процесс восстановления (restore). Надо сказать, что процесс восстановления занимает примерно в 3 раза больше времени, чем резервное копирование, и для больших баз проверку на восстановление ежедневно делать затруднительно, так как это может прервать работу пользователей на несколько часов, т. е. ночного перерыва может и не хватить. Конечно, организациям, имеющим базы данных такого большого объема лучше, не экономить "на спичках" и выделить отдельный компьютер для этих целей.

В том случае, если сервер должен функционировать при значительной нагрузке по 24 часа 7 дней в неделю, можно воспользоваться механизмом SHADOW для снятия "моментальных" снимков с базы данных и дальнейших операций по резервному копированию уже с моментальной копией. Подробно

процесс резервного копирования и восстановления баз данных описан в главе "Резервное копирование и восстановление базы данных из резервной копии".

При создании резервной копии и затем при восстановлении базы данных из этой копии происходит пересоздание всех данных в базе данных. Этот процесс (backup/restore, или коротко – b/r) способствует исправлению большинства нефатальных ошибок в базе данных, связанных с повреждениями жесткого диска, выявляет проблемы с целостностью данных в базе, очищает базу данных от "мусора" (старых версий и фрагментов записей, незавершенных транзакций), значительно уменьшает размер базы данных. Можно сказать, что регулярный b/r – залог здоровья базы данных InterBase. Если база данных рабочая, то рекомендуется производить b/r еженедельно. Правда, есть свидетельства о базах данных InterBase, которые интенсивно используются по несколько лет без backup/restore. Тем не менее для собственного спокойствия все-таки желательно производить эту процедуру, тем более что ее легко можно автоматизировать.

Если по каким-то причинам невозможно часто производить процесс backup/restore (особенно restore), то можно воспользоваться инструментом для проверки и восстановления баз данных gfix, который позволяет провести проверку и восстановление многих ошибок без b/r.

Инструмент командной строки gfix

Для проверки и восстановления базы данных используется инструмент gfix. Помимо этого, gfix также может выполнять различные действия по управлению базой данных: менять диалект базы данных, устанавливать и снимать режим работы "только чтение", устанавливать размер кеша для конкретной базы данных, а также некоторые важные функции, с которыми можно подробно ознакомиться в [4].

Инструмент gfix выполняется в режиме командной строки и имеет следующий синтаксис:

```
gfix [ options] db_name
```

Options – это набор опций для выполнения gfix, a db_name – имя базы данных, над которой будут производиться операции, определенные набором опций. В таблице 4.11 представлены опции gfix, относящиеся к "починке" базы данных:

Таблица 4.11. Опции инструмента gfix для восстановления базы данных

Опция	Описание опции
–f[ull]	Используется в сочетании с –v и означает, что необходимо проверять все фрагменты записей
–i[gnore]	Заставляет gfix игнорировать ошибки контрольных сумм во время проверки или очистки базы данных
–m[end]	Отмечает поврежденные записи как недоступные, в результате чего они удалятся при последующем backup/restore. Опция применяется во время подготовкой поврежденн базы данных к b/r
–n[o_update]	Используется в сочетании с –v для read-only-проверки базы данных, без исправления повреждений

Опция	Описание опции
–pas[sword]	Позволяет задать пароль при подключении к базе данных
–user name	Позволяет задать имя пользователя при подключении к базе данных
–v[alidate]	Задает проверку базы данных, в ходе которой обнаруживаются ошибки в структуре
-m[ode]	Устанавливает режим записи для базы данных – только для чтения или чтение/запись. Этот параметр может принимать два значения: read_write или read_only
–w[rite] {sync | async}	Включает/выключает режимы синхронной/асинхронной записи (forced writes) в базу данных: sync – включить синхронную запись (FW ON); async – включить асинхронную запись (FW OFF)

Вот несколько типичных примеров использования gfix:

```
gfix – w sync firstbase.gdb
```

В этом примере мы устанавливаем для нашей тестовой базы данных firstbase.gdb режим синхронной записи (FW ON).

```
gfix –v –full firstbase.gdb
```

В этом примере мы запускаем проверку нашей тестовой базы данных (опция -v), причем указываем, что необходимо проверить также фрагменты записей (-full).

Конечно, назначать различные опции для процесса проверки и восстановления удобнее с помощью какого-нибудь графического инструмента администрирования, но мы будем рассматривать функции восстановления базы данных с точки зрения применения именно инструментов командной строки. Эти инструменты входят в поставку InterBase, и можно быть уверенным, что они будут вести себя одинаково во всех ОС, поддерживаемых InterBase. Не менее важен тот факт, что они всегда окажутся под рукой. Кроме того, существующие инструменты, позволяющие выполнять администрирование баз данных с клиентского компьютера, используют для этого Services API, которое не поддерживается серверами InterBase с архитектурой Classic. То есть вы сможете использовать сторонние продукты только с серверами архитектуры SuperServer.

Восстановление поврежденной базы данных

Предположим, что наша база данных содержит ошибки и нам необходимо, во-первых, проверить наличие этих ошибок, во-вторых, попытаться исправить эти ошибки. Порядок действий при этом рекомендуется соблюдать следующий.

Останавливаем сервер InterBase, если он еще работает, и делаем копию файла или файлов базы данных. Все действия по восстановлению следует производить только над копией базы данных, так как выбранная стратегия может привести к неудаче и придется начать процедуру восстановления снова – с начального состояния базы данных.

После создания копии произведем полную проверку базы данных (с проверкой фрагментов записей), для чего необходимо выполнить следующую команду:

```
gfix –v –full corruptbase.gdb – user SYSDBA –password
<ваш_пароль>
```

В данном случае corruptbase.gdb – это копия поврежденной базы данных. Команда проверит базу данных на предмет повреждения любых структур и выведет список неразрешенных проблем. Если обнаружатся такие ошибки, то нам придется пометить поврежденные данные для удаления и подготовиться к процессу backup/restore, используя следующую команду:

```
gfix –mend corruptbase.gdb –user SYSDBA – password <ваш_пароль>
```

После выполнения этой команды следует проверить, остались ли ошибки в базе данных, для чего необходимо вновь запустить gfix с опциями –v –full, а после того, как он отработает, произвести резервное копирование базы данных:

```
gbak –b –v –ig – user SYSDBA –password <ваш_пароль> corrupt-
base.gdb corruptbase.gbk
```

Эта команда произведет резервное копирование базы данных (об этом говорит опция -b), при этом будут выводиться подробные сведения о ходе backup (опция –v), причем ошибки, связанные с контрольными суммами, будут игнорироваться (опция –ig). Подробнее об опциях инструмента командной строки gbak можно посмотреть в главе "Резервное копирование и извлечение базы данных из резервной копии" этой части.

В случае ошибок с backup следует запустить его в другой конфигурации:

```
gbak –b –v –ig –g – user SYSDBA –password <ваш_пароль>
corruptbase.gdb corruptbase.gbk
```

где опция –g запретит сборку мусора во время резервного копирования. Часто это помогает решить проблему с backup. Но бывает, что и такого сочетания опций недостаточно для успешного завершения процесса backup. Тогда следует добавить в команду резервного копирования опции –inactive и –one_at_a_time, которые деактивируют индексы в создаваемой из backup-копии базы данных и производят подтверждение (commit) данных для каждой таблицы соответственно.

Также бывает возможным сделать резервную копию базы данных, если перед этим предварительно перевести базу данных в режим "только чтени" (read-only). Такой режим препятствует записи любых изменений а базу данных и иногда помогает осуществить backup поврежденной базы данных. Для перевода базы данных в режим "только чтение" следует воспользоваться следующей командой:

```
gfix –m read_only –user SYSDBA –password masterkey
Disk:\Path\file.gdb
```

После этого необходимо вновь попытаться сделать backup базы данных с приведенными выше параметрами.

Спасение данных из поврежденной базы данных

Возможно, что все вышеприведенные действия не приведут к восстановлению базы данных. Это означает, что база серьезно повреждена и либо совсем не подлежит восстановлению как единое целое, либо для ее восстановления понадобится приложить значительное количество усилий. Например, можно вручную осуществить модификацию системных метаданных, воспользоваться недокументированными функциями и т. д. Это очень тяжелая, длительная и неблагодарная работа с сомнительными шансами на успех, поэтому по возможности надо избегать ее и пользоваться другими способами. Если поврежденная база данных открывается и позволяет производить операции чтения и модификации хотя бы над частью данных, то нужно воспользоваться этой возможностью и спасать данные путем перекачки их в новую базу, а со старой базой распрощаться навсегда.

Итак, чтобы приступить к переносу данных из старой базы данных, надо сначала создать базу-приемник. Если база данных устоявшаяся (метаданные давно не менялись), то для создания базы данных-приемника можно воспользоваться какой-нибудь старой backup-копией, из которой можно извлечь метаданные (для этого лучше всего воспользоваться какой-нибудь утилитой администрирования). На основе этих метаданных необходимо создать базу данных-приемник и приступить к перекачиванию данных. В принципе главное – это вытащить данные из поврежденной базы данных, а размещение этих данные в новой базе является задачей менее трудоемкой и сложной, даже если придется восстанавливать структуру базы данных "по памяти".

При извлечении данных из таблиц следует пользоваться следующим алгоритмом действий:

1. Сначала нужно попытаться выполнить SELECT * FROM tableN. Если это прошло нормально, то вы можете сохранить полученные данные во внешнем источнике. Для этой цели хорошо подходит сохранение данных в скрипт (такую функцию предоставляют почти все графические утилиты администрирования), если только таблица не содержит BLOB-полей. Если в таблице есть BLOB-поля, то данные из них необходимо сохранять в другую базу данных с помощью программы-клиента, которая будет исполнять роль посредника. Возможно, вам придется написать эту тривиальную программу специально для целей восстановления данных.
2. Если запрос на выборку всех данных "не прошел", то необходимо удалить все индексы и повторить запрос. В сущности, индексы можно удалить у всех таблиц с самого начала восстановления – они больше не понадобятся. Естественно, если у вас нет структуры метаданных, идентичной поврежденной, необходимо вести протокол всех действий, которые вы проделываете над поврежденной базой данных-источником.
3. Если после удаления индексов не удается прочитать все данные из таблицы, то можно попробовать делать выборки с интервалами по первичному ключу, – т. е. выбирать определенные диапазоны данных, например:
```
SELECT * FROM tableN WHERE field_PK>=0 and field_PK <=10000
```

Здесь field_PK – поле, которое исполняет роль первичного ключа. Так как InterBase имеет страничную организацию данных, то выборка диапазонов значений может оказаться достаточно эффективной, хотя это и кажется чем-то вроде шаманства. Тем не менее это работает, поскольку при этом мы можем исключить из выборки данные с поврежденных страниц, а остальные успешно прочитать. Вы можете вспомнить наш тезис о том, что в SQL нет определенного порядка хранения записей. Да, действительно, никто не гарантирует, что неупорядоченная выборка при повторных запусках вернет нам записи в одинаковом порядке, но тем не менее физически записи хранятся внутри базы данных в определенном внутреннем порядке. Очевидно, что сервер не станет "перетасовывать" записи просто ради следования букве SQL-стандартов. Этим внутренним порядком можно попытаться воспользоваться, извлекая данные из поврежденной базы данных (подробнее о страницах данных и их взаимосвязях см. ниже в главе "Структура базы данных InterBase"). Виталий Бармин, один из опытных российских InterBase-разработчиков, сообщал о том, что таким образом ему удалось извлечь из сильно попорченной базы данных, в которой было множество поврежденных страниц, до 98% информации!

Таким образом, данные из поврежденной базы данных необходимо перенести в новую базу или же просто во внешние источники данных, наподобие sql-скриптов. При копировании данных обратите внимание на значения генераторов в поврежденной базе – их необходимо сохранить для возобновления корректной работы в новой базе данных. Если у вас отсутствует полная копия метаданных, то необходимо, хотя бы частично, извлечь из поврежденной базы данных тексты хранимых процедур, триггеров, ограничений и определения индексов.

Восстановление "безнадежных" баз данных. InterBase Surgeon

В общем, восстановление базы данных может быть очень хлопотным и тяжелым делом, поэтому никогда не бывает лишним сделать резервную копию базы данных, чтобы не заниматься восстановлением поврежденных данных. Тем не менее, даже если это и случилось, не стоит отчаиваться – выход можно найти даже в самых, казалось бы, безнадежных ситуациях, и сейчас мы рассмотрим два таких случая.

Первый случай представляет собой классическую проблему – это невосстанавливающаяся резервная копия из-за наличия NULL-значений в столбце с ограничениями NOT NULL, которую пустили на восстановление прямо поверх рабочего файла. Рабочий файл затерся, процесс восстановления (restore) прервался из-за ошибки, и мы получили в результате необдуманных действий большой кусок бесполезных данных вместо резервной копии, который лежит на диске и, кажется, не поддается восстановлению. Но выход все же был найден. В той реальной ситуации, в которой мы нашли данное решение, программист сумел вспомнить, на какую именно таблицу и на какой столбец было наложено злополучное ограничение NOT NULL. Файл резервной копии был загружен в шестнадцатеричный редактор, и там путем поиска было найдено сочетание байтов, соответствующее определению этого столбца. После многочисленных

экспериментов выяснилось, что ограничение NOT NULL добавляет единичку "где-то рядом" с именем столбца. Прямой правкой в HEX-редакторе эта единичка была исправлена на 0, и резервная копия была восстановлена. Программист отделался легким испугом и навсегда запомнил, как правильно организовывать процесс резервного копирования и восстановления из резервной копии.

Следующий случай казался совершенно безнадежным, однако выход все же был найден.

Ситуация сложилась катастрофическая. База данных "сломалась" на этапе расширения из-за нехватки дискового пространства. InterBase записал на странице учета страниц число страниц, превышающее ее реальный размер. Более того, на страницах учета страниц, похоже, также образовались повреждения. В результате база данных вообще не открывалась ни средствами администрирования, ни утилитой gbak, а при попытке подключиться к базе данных появлялась ошибка "Unexpected end of file". При запуске утилиты gfix происходили весьма странные вещи: программа попросту входила в бесконечный цикл. Сервер в момент работы gfix с огромной скоростью (около 100 Кбайт/с) писал ошибки в лог (файл interbase.log). В результате файл лога очень быстро заполнял все свободное дисковое пространство, и пришлось даже написать программу, которая по таймеру уничтожала неимоверно распухающий лог. Этот процесс продолжался довольно долго – gfix работал более 16 ч без каких-либо видимых результатов.

Лог наполнялся ошибками вида "Page XXXX doubly allocated". В исходных кодах InterBase (в файле val.c) есть скупое описание этой ошибки. Оно гласит, что данная ошибка возникает в случае, когда одна и та же страница данных используется дважды. Очевидно, что эта ошибка, как и зацикливание, являлась следствием разрушения критически важных страниц в базе данных. В результате после нескольких дней безуспешных экспериментов попытки восстановить данные стандартными (документированными) способами были оставлены. Поэтому пришлось перейти к низкоуровневому анализу данных, хранящихся в поврежденной базе данных, точнее, в том неупорядоченном массиве данных, который от нее остался.

Автором идеи о том, как извлечь информацию из подобных "безнадежных" баз данных является Александр Козельский, начальник отдела информационных технологий компании East View Publications Inc. (www.eastview.com).

Метод восстановления, полученный в результате исследований, основывался на том факте, что база данных имеет страничную организацию, а данные из каждой таблицы сгруппированы по страницам данных. Каждая страница данных содержит идентификатор таблицы, для которой она хранит данные.

Особенно было важно восстановить данные из нескольких критичных таблиц. Существовали данные из таких же таблиц, полученные из старой резервной копии, которая отлично работала и могла служить образцом.

База данных-образец была загружена в редактор шестнадцатеричных кодов, затем был произведен поиск образцов тех данных, которые нас интересовали. Шестнадцатеричное представление этих данных было скопировано в буфер, а затем в редактор загрузили остатки поврежденной базы данных. В поврежденной базе данных была найдена последовательность байтов, соответствующая образцу, и была проанализирована страница, на которой нашлась эта последова-

тельность. Сначала было определено начало страницы, что оказалось несложно, поскольку размер файла базы данных кратен размеру страницы данных. Найти начало страницы также оказалось возможным, для этого пришлось поделить номер текущего байта на размер страницы – 8192 байта, а затем округлить результат до целых (в результате получился номер текущей страницы) и умножить полученное число на размер страницы. Таким образом, был найден номер байта, соответствующий началу текущей страницы. Проанализировав заголовок, определили тип страницы (для страниц с данными тип равен пяти – см. файл ods.h из комплекта исходных кодов InterBase, а также ниже главу "Структура базы данных InterBase"), а также идентификатор нужной таблицы.

Дальше была написана программа, которая "просматривала" всю базу данных, собирала все страницы для нужной нам таблицы в единый кусок и сбрасывала его в файл.

Получив таким образом "выжимку" только тех данных, которые были нужны в первую очередь, приступили к анализу содержимого отобранных страниц. InterBase активно использует сжатие данных для экономии места. Например, строку типа VARCHAR, содержащую строку "ABC", он хранит в виде последовательности таких значений: длины строки (2 байта, в нашем случае это 00 03), затем самих символов, а потом контрольной суммы. Пришлось написать анализатор строк, а также других типов данных, который преобразовывал специализированное шестнадцатеричное представление данных в "нормальный" вид.

Применив такой метод "рукопашного" разбора содержимого базы данных, удалось извлечь около 80% информации из нескольких критически важных таблиц.

Позже на основе полученного опыта Олегом Кульковым и Алексеем Ковязиным, одним из авторов этой книги, была разработана утилита *IBSurgeon* (т. е. InterBase Хирург), которая осуществляет прямой доступ к базе данных, минуя ядро InterBase-сервера, и позволяет напрямую читать и корректным образом интерпретировать данные внутри базы данных InterBase.

Используя InterBase Surgeon, удается распознать причины повреждения и восстановить до 90 %, казалось бы безнадежных баз данных, которые не открываются с помощью InterBase и не "лечатся" стандартными методами.

Скачать эту программу можно с сайта поддержки данной книги www.InterBase-world.com, а также с официального сайта программы www.ibsurgeon.com.

Статистика в InterBase

Статистика – одно из изобретений человечества, которое позволяет количественно оценить динамику развития каких-либо процессов и принимать важные решения на основе цифр, а не только интуиции. InterBase предоставляет большое количество статистических данных, многие из которых могут быть использованы для принятия решений, способствующих оптимизации работы сервера. В этой главе будут рассмотрены виды статистической информации InterBase и способы ее получения и интерпретации.

Статистика в InterBase бывает двух видов – статистика базы данных и статистика сервера. Сначала мы рассмотрим статистику базы данных.

Статистика базы данных InterBase

Из базы данных InterBase можно извлечь следующие виды статистики, различающиеся тем, откуда была получена информация:

- Статистика заголовочной страницы (header page information). Это информация о глобальных свойствах всей базы данных, которая содержится на заголовочной странице каждой базы данных.
- Статистика страницы протокола базы данных (database log page information). Это информация не используется в современных версиях InterBase 6.x и рассматриваться здесь не будет, хотя с помощью утилиты gstat можно получить представление о том, как выглядела эта статистическая информация.
- Статистика страниц данных (data page information). Это информация о таблицах, содержащихся в базе данных, которая включает в себя номера страниц данных и страниц указателей для данной таблицы, степень заполнения страниц и другую важную информацию.
- Статистика индексов (index information). Это информация об индексах в базе данных, включающая имя индекса, его глубину, среднюю длину ключей, процент заполнения страниц индексов и т. д.
- Статистика системных таблиц (system relations information). Это информация о системных таблицах и индексах. Она аналогична статистике для обычных таблиц и индексов.

Мы подробно рассмотрим все эти виды статистики, но сначала остановимся на способе ее получения.

Получение статистики

Существует много способов получить статистику. Почти все универсальные инструменты, перечисленные в приложении "Инструменты администратора и разработчика InterBase", позволяют получить статистику базы данных с помощью нескольких нажатий мыши, однако часто случается так, что нужных инструментов не оказывается под рукой; поэтому мы рассмотрим, как получить результат, пользуясь только стандартными средствами. К таковым относится утилита командной строки gstat, которая входит в стандартную поставку InterBase 6.x и его клонов и позволяет получить все вышеперечисленные виды

статистики. Правда, есть важное ограничение – gstat должна выполняться на том же компьютере, где находится сервер InterBase, т. е. удаленное получение статистики при помощи gstat невозможно.

Формат использования gstat следующий:

```
gstat [options] database
```

Здесь database – имя и путь к базе данных, из которой будет извлекаться статистика, а [options] – набор опций, которые определяют, какую информацию надо получить. Опции утилиты gstat описаны в таблице 4.12:

Таблица 4.12. Опции gstat

Опция	Описание опции
–all	Опция выбирается по умолчанию – приводит к извлечению статистики по страницам данных и индексам
–data	Извлекает статистику по страницам данных всех пользовательских таблиц в базе данных
–header	Извлекает только статистику заголовочной страницы
–index	Извлекает статистику по индексам в базе данных
–log	Извлекает только статистику о страницах протокола
–pas[sword]	Пароль пользователя, который запускает gstat для получения статистики
–system	Извлекает статистику по системным таблицам и индексам
–user name	Пользователь InterBase, который запускает gstat для получения статистики. Только владелец базы данных или системный администратор SYSDBA может запускать gstat для получения статистики
–z	Печатать версию gstat

Помимо использования утилиты gstat, статистику всегда можно получить, применяя Services API, который реализован во всех версиях InterBase 6.x и его клонов в варианте SuperServer, а также в клоне Yaffil Classic Server. Воспользоваться Services API можно как на низком уровне, так и посредством специализированных библиотек доступа к InterBase, таких, как FIBPlus и IBX. При использовании Services API нет ограничения на то, чтобы клиент, запрашивающий статистику, обязательно находился на компьютере-сервере.

В наших примерах мы будем использовать утилиту gstat, как наиболее надежный и стандартный способ получения статистики. Пример получения полной статистики базы данных выглядит следующим образом:

```
gstat –all –user SYSDBA –password masterkey
C:\Database\firstbase.gdb
```

При этом будет выведена как общая информация о базе данных, так и подробная информация о таблицах в базе данных. Надо сказать, что результатом выполнения этой команды даже для небольшой базы данных будет довольно обширный отчет. Давайте более подробно рассмотрим информацию, извлекаемую утилитой gstat.

Информация заголовочной страницы (Database header)

Заголовочная страница содержит важную информацию о базе данных в целом. Часть информации является статичной и записывается при создании базы данных, часть – меняется в зависимости от происходящих с базой данных изменений. Запуск утилиты gstat с ключом –all приводит к выводу всей статистической информации базы данных, при этом информация с заголовочной страницы выводится первой. Чтобы прекратить вывод статистики сразу после вывода этой информации, следует при запуске gstat указать только ключ -h. Пример информации с заголовочной страницы приводится ниже:

```
Database "C:\Database\EVP.GDB"
Database header page information:
        Flags                   0
        Checksum                12345
        Generation              1100521
        Page size               8192
        ODS version             8.2
        Oldest transaction      1084640
        Oldest active           1100476
        Oldest snapshot         1100476
        Next transaction        1100478
        Bumped transaction      1
        Sequence number         0
        Next attachment ID      0
        Implementation ID       8
        Shadow count            0
        Page buffers            0
        Next header page        0
        Database dialect        1
        Creation date           Dec 19, 2001 21:30:59
        Attributes              force write

    Variable header data:
        Shared Cache file:
        Sweep interval:         20000
        *END*
```

В первой строке содержится информация о расположении файла базы данных. Далее идет блок информации, озаглавленный Database header page information.

Flags

Первой строкой в нем идет параметр Flags. Это набор флагов, определяющий важные особенности поведения базы данных. Возможные значения флагов, взятые из файла ods.h, описывающего структуру базы данных (On-disk structure – см. ниже главу "Структура базы данных InterBase"), приведены ниже в табл. 4.13.

Таблица 4.13. Флаги файла базы данных

Значение флага (десятичное и шестнадцатеричное)	Расшифровка его значения
0x1 1	Файл является активным Shadow-файлом
0x2 2	Режим синхронного чтения-записи включен (forced write on)
0x4 4	Краткосрочное журналирование
0x8 6	Долгосрочное журналирование
0x10 8	Не вычислять контрольные суммы
0x20 16	Не резервировать место для версий файлов
0x40 62	Запретить применение совместно используемого кеш-файла
0x80 128	База данных остановлена
0x100 256	В базе данных используется SQL диалект 3
0x200 512	База данных только для чтения. Если флаг не установлен, то допустимы как чтение, так и запись

Флаги устанавливаются только с помощью специальных инструментов вроде gfix, изменять флаги с помощью других инструментов опасно – это может привести к порче базы данных.

Надо сказать, что при получении статистики показывается, что значение параметра Flags всегда равно нулю, вне зависимости от установленных флагов. Дело в том, что расшифровка части флагов производится ниже – в параметрах Database Dialect и Attributes.

Checksum

Второй строкой идет параметр checksum, т. е. контрольная сумма. Контрольная сумма имеется как на заголовочной странице, так и на любой другой странице базы данных. Однако в современных версиях InterBase (для ODS старше 9.x) она не используется и ее значение всегда равно 12345. Не очень полезный параметр.

Generation

Третья строка – это generation ("поколение" в переводе с английского). Это счетчик, который увеличивается на единицу всякий раз, когда заголовочная страница записывается на диск. Тоже мало полезный параметр.

Page size

Размер страницы базы данных, исчисляется в байтах. Параметр, который устанавливает основополагающее свойство базы данных – размер ее страницы. Все файлы одной базы данных состоят из страниц одинакового размера, кото-

рый устанавливается при создании базы данных и при восстановлении базы данных из резервной копии. Впрочем, восстановление можно считать частным случаем создания базы данных. Множество важнейших параметров сервера зависят от размера страницы – например, кеш базы данных. Рекомендуют создавать базу данных с размером страниц не менее 4096 байт, а лучше 8192 или 16384 байта. Последнее, правда, доступно лишь в Firebird и Yffil (см. ниже главу "Структура базы данных InterBase").

ODS version

Версия On-Disk structure – структуры базы данных InterBase. Представляет собой два числа, разделенные точкой. "Целая" часть – это основная версия ODS, которая зависит от версии сервера, создавшего данную базу данных. Главная версия определяет основные возможности работы с базой данных, и ее значение присваивается при создании (восстановлении) базы данных.

"Дробная" часть (после точки) – это минорная версия ODS, которая может меняться (точнее, увеличиваться) в течение жизни базы данных в зависимости от того, под управлением какой версии сервера работают с этой базой данных.

Oldest transaction

Параметр Oldest transaction показывает идентификатор старейшей заинтересованной транзакции в базе данных. Значение этого параметра часто сравнивается с Next transaction. Разница этих значений является важным показателем "здоровья" базы данных и косвенно указывает на положение дел со сборкой "мусора" в базе данных. Подробно "Oldest transaction" рассмотрена в главе "Транзакции. Параметры транзакций" (ч. 1).

Oldest active и Oldest snapshot

Параметр Oldest active – идентификатор старейшей активной транзакции. Обычно его значение близко к значению Next transaction. Параметр Oldest snapshot в документации по InterBase 6 не описывается. Однако Анн Харрисон любезно разъяснила смысл этого параметра. Дело в том, что только транзакции с уровнем изоляции SNAPSHOT вызывают появление мусорных версий записей (garbage) в базе данных, и этот параметр показывает номер последней транзакции snapshot, которая влияет на процесс сборки мусора.

Next transaction

Идентификатор следующей транзакции, которая будет запущена сервером.

Bumped transaction

Этот параметр более не используется в InterBase. Он является наследием тех версий InterBase, которые использовали так называемый Write-ahead log.

Sequence number

Порядковый номер файла базы данных. Для базы данных, состоящей из одного файла, он всегда равен нулю. Второй файл в базе данных будет иметь номер 1 и т. д. В документации ошибочно написано, что это номер заголовочной страницы, однако анализ исходных кодов InterBase и эксперименты показывают, что это не так.

Next attachment ID

Идентификатор следующего подсоединения к базе данных. Вероятно, всегда равен нулю.

Implementation ID

Параметр Implementation ID определяет, под какой ОС была создана база данных. Табл. 4.14, взятая из Operation Guide, из комплекта документации InterBase 6.x, дает нам информацию обо всех возможных ОС, поддерживаемых в InterBase. В ней содержатся ОС, под которые хоть когда-то выпускался InterBase. Учитывая, что одна из версий InterBase (мы не имеем точной информации о том, какая эта версия) применяется на танках "Abrams", то, вероятно, этот список может быть и неполным. Хотя Implementation ID=3, 6, 9 и 14 наводят на размышление. Тем не менее вы, вероятно, согласитесь с тем, что список все равно получается внушительный.

Таблица 4.14. ОС, поддерживаемые InterBase (все версии)

Значения Implementation ID	ОС
1	Apollo
2	Sun, HP 9000, IMP Delta, NeXT
3	Reserved
4	VMS
5	VAX Ultrix
6	Reserved
7	HP 900
8	OS/2, Windows NT, Novell NetWare
9	Reserved
10	RS 6000
11	Data General AviiON
12	HP M PE/XL
13	Silicon Graphics Iris
14	Reserved
15	DEC O SF/1
16	Windows 95/98/Me

Shadow count

Число файлов Shadow, которые определены для данной базы данных. Как известно, файлы Shadow представляют собой зеркальные подобия основной базы данных. Ранее предназначенные для предохранения базы данных от неожиданной поломки жесткого диска, теперь они в основном используются в целях резервного копирования – для снятия мгновенных снимков базы данных (подробнее см. в этой части, в главе "Резервное копирование базы данных и восстановление из резервной копии").

Page buffers

Размер кеша базы данных, исчисляемый в страницах. Определяет число страниц, которое может быть одновременно размещено в кеше. Размер кеша является важнейшим параметром, влияющим на производительность операций чтения ввода-вывода базы данных (см. ниже главу "Оптимизация работы InterBase"). Надо отметить, что установка размера кеша на уровне базы данных, отражаемая данным параметром, имеет значение только для InterBase SuperServer.

Next header page

Номер следующей заголовочной страницы. Всегда равен нулю. Собственно говоря, любая страница в базе данных имеет ссылку на номер следующей страницы такого же типа (см. ниже главу "Структура базы данных InterBase"), но так как заголовочная страница всегда единственная в каждом файле базы данных, то у нее эта ссылка обнулена.

Database dialect

Диалект, используемый в базе данных. Может принимать значения 1 или 3. Диалект 2 могут иметь только клиенты InterBase (см. в этой части главу "Миграция").

Creation date

Дата создания базы данных.

Attributes

Атрибуты базы данных. Могут иметь следующие значения: force write, no_reserve и shutdown. Очевидно, что значения этих атрибутов соответствуют флагам, хранящимся в параметре Flags.

Далее идет информация, озаглавленная Variable header data (переменные данные заголовочной страницы):

Shared Cache file

Параметр более не используется в InterBase.

Sweep interval

Параметр устанавливает интервал между старейшей активной и следующей транзакциями (OIT и Next), после которого начинается сборка "мусора". Более подробно об этом можно прочитать в главах "Оптимизация работы InterBase" (ч. 4) и "Транзакции. Параметры транзакций" (ч. 1).

Помимо рассмотренных параметров, информация в заголовочном файле содержит информацию о вторичных файлах базы данных – имя, размер и расположение.

Теперь можно перейти к рассмотрению других видов статистической информации, получаемых из базы данных InterBase.

Информация страниц данных

Если при запуске gstat укажем ключи –all или –data, то получим информацию обо всех пользовательских таблицах в базе данных. Информация для каждой таблицы выглядит так:

```
CUSTOMER (33)
    Primary pointer page: 129, Index root page: 130
    Data pages: 664, data page slots: 749, average fill: 90%
    Fill distribution:
         0 - 19% = 4
        20 - 39% = 14
        40 - 59% = 36
        60 - 79% = 55
        80 - 99% = 555
```

Здесь CUSTOMER –имя таблицы; 33 – идентификатор таблицы (соответствует RDB$RelationID в системной таблице RDB$Relations); Primary pointer page – это номер первой страницы указателей в базе данных для данной таблицы; Index root page – номер первой страницы указателей для индексов этой таблицы; Data pages – число страниц данных, занимаемых данной таблицей; Data page slots – число указателей на страницы данных; average fill – процент использования страниц под хранящиеся данные; Fill distribution – табл. показывающая, какое количество страниц как заполнено. Например, первая строка означает, что 4 страницы заполнены на 0–19 %, а последняя – что 555 страниц, занятых этой таблицей, заполнены на 80–99 %.

Надо сказать, что, несмотря на обилие узкотехнических терминов в получаемой статистике (которые рассмотрены ниже в главе "Структура базы данных InterBase"), добываемые данные все же могут быть использованы для оптимизации базы данных даже неспециалистами. Например, если данные на страницах слишком разрежены, то надо задуматься о структуре таблицы. Возможно, требуется разнести поля этой таблицы по нескольким таблицам, чтобы они плотнее размещались на диске, в результате чего можно получить некоторое ускорение операций чтения и записи.

Статистика страниц индексов

Чтобы получить статистику по страницам индексов, необходимо либо указать при запуске gstat ключ –index, либо запустить с ключом –all. Статистика по одному индексу выглядит следующим образом:

```
CUSTOMER (33)
Index CONTACT_IDX (5)
        Depth: 2, leaf buckets: 32, nodes: 20005
        Average data length: 1.00,
        total dup: 17584, max dup: 12096
        Fill distribution:
             0 - 19% = 0
            20 - 39% = 0
            40 - 59% = 22
            60 - 79% = 4
            80 - 99% = 6
```

Здесь CUSTOMER – имя таблицы, для которой создаются индексы. CONTACT_IDX – это имя индекса, а 5 – порядковый номер индекса для данной таблицы (нумерация начинается с нуля). Самым важным параметром в получаемой информации является глубина индекса – Depth. Она определяет, сколько веток в "дереве" индекса необходимо пройти, чтобы вычислить нужную запись (подробнее см. в главе "Индексы" (ч. 1)). Важно, что глубина индекса не была больше трех, так как в этом случае эффект от использования индекса невелик. Если она больше, то следует увеличить размер страницы данных в базе данных. Остальные параметры означают следующее: leaf buckets – число страниц на концах дерева ("листьев"); nodes – полное число узлов в индексе; Average data length – средняя длина ключей в индексе. Параметры total dup и max dup характеризуют число повторений записей в индексе. Таблица Fill distribution аналогична по своему назначению такой же таблице, входящей в состав информации о страницах данных.

Статистика базы данных дает возможность точнее диагностировать проблемы, возникающие в процессе эксплуатации, позволяет произвести тонкую настройку производительности и является неотъемлемой частью профилактических процедур, производимых над базой данных.

Статистика InterBase-сервера

Часто разработчики программ под InterBase желают получить информацию о поведении сервера во время выполнения запросов – использовании процессорного времени, загрузке памяти и т. д. Чтобы удовлетворить эти запросы, в InterBase и его клонах реализована функция InterBase API isc_database_info(), возвращающая необходимую информацию. Параметры этой функции перечислены в [4, гл. 8]. Очевидно, что писать каждый раз, когда потребовалась статистика, специальную программу с вызовами InterBase API неудобно, да и не нужно, потому как множество подобных высококачественных продуктов уже существует.

Одним из таких продуктов является MiTeC InterBase Performance Monitor (IBPM). Это свободно распространяемая программа, автором которой является Михаэль Матл (Michal Mutl). Этот монитор работает на всех ОС Windows, начиная с Windows 95. Эта программа отслеживает параметры InterBase-сервера, представленные в табл. 4.15.

Таблица 4.15. Параметры производительности сервера

Параметра	Описание параметра
Memory and CPU usage	Использование памяти и процессора
Reads from memory buffer cache	Число чтений из кеша в памяти с момента запуска InterBase-сервера
Writes to memory buffer cache	Число записей в кеш-память с момента запуска InterBase-сервера
Reads from database	Число чтений из базы данных с момента запуска текущего соединения
Writes to database	Число записей в базы данных с момента запуска текущего соединения
Active users	Число активных пользователей
Server log	Если сервер поддерживает Services API для статистики, то можно получить протокол работы сервера
Allocated pages	Число распределенных страниц в базе данных
Number of buffers	Размера кеша
Removals of a version of a record	Число удалений версий записей с момента текущего соединения
Removals of fully mature records	Число удалений записей с момента текущего соединения
Removals of a record and all of its ancestors	Число удалений записей и всех предков
Reads done via index	Количество чтений, осуществленных с помощью индекса (вычисляется для каждой таблицы с момента текущего соединения)
Sequential table scans	Количество последовательных проходов по таблицам
Database updates	Число обновлений в базе данных с момента текущего соединения
Database inserts	Число вставок в базу данных с момента текущего соединения
Database deletes	Число удалений в базе данных с момен-

Параметра	Описание параметра
	та текущего соединения
Local SQL Monitor (for apps based on FIBPlus)	Для приложений, написанных на FIBPlus, есть возможность отслеживать SQL-запросы, направляемые к базе данных

Как видите, достаточно большое поле для изучения поведения InterBase-сервера. MiTeC InterBase Performance Monitor не требует специальной установки – достаточно, чтобы на компьютере, где он запускается, был установлен клиент InterBase. Удобный интерфейс поможет легко разобраться с использованием данной программы.

Помимо вышеперечисленных возможностей MiTeC IBPM может также извлекать статистику базы данных, которую можно получать с помощью утилиты gstat.

Статистика по блокировкам

InterBase использует механизм блокировок, чтобы организовывать совместную работу многих пользователей с одной базой данных. Изучение статистики по блокировкам позволяет регулировать настройки механизма блокирования. Подробнее об этом написано в приложении "Параметры конфигурационного файла InterBase". Для получения такой статистики нужно использовать утилиту iblockpr (для Windows, а для Linux – gds_lock_print), которая предназначена специально для получения данных о блокировках. Обычно эта утилита располагается в подкаталоге BIN в основном каталоге InterBase.

Формат ее запуска и применяемые подробно изложены в [4, гл. 8], поэтому мы не будем здесь его приводить, тем более что область применения подобной статистики достаточно узка: ее анализ может производиться при тонкой настройке конфигурации сервера. В 90 % случаев изменять параметры механизма блокировок не требуется.

Заключение

Статистику InterBase-сервера и базы данных необходимо проанализировать на этапе тестирования приложения баз данных, чтобы выявить возможные "узкие места" в производительности, а также во время профилактических процедур по обслуживанию базы данных после возникновения сбоев или каких-либо проблем.

Оптимизация работы InterBase

В этой главе мы рассмотрим, какое аппаратное обеспечение следует использовать для InterBase, а также выясним возможности оптимизации InterBase-сервера и его клонов, разберем параметры конфигурационных файлов InterBase и их назначение.

Выбор аппаратного обеспечения для InterBase

Аппаратное обеспечение ("железо"-на компьютерном жаргоне) – это компьютер-сервер и его компоненты, сетевое оборудование, а также рабочие станции, на которых будут выполняться клиентские программы, использующие InterBase. Более всего внимания мы уделим компьютеру-серверу, но также дадим несколько рекомендаций относительно остальных компонентов.

Конечно, разные задачи требуют различного аппаратного обеспечения. Поэтому здесь мы будем рассматривать преимущества и недостатки аппаратного обеспечения исходя, из того, что выбор СУБД для задачи уже позади, т. е. твердо выбран InterBase. Это позволит нам избежать замечаний вроде: "При таких объемах данных вам лучше еще раз подумать об Oracle".

Разумеется, выбор аппаратного оборудования нельзя рассматривать как оптимизационный процесс, который целиком относится только к InterBase, однако часто бывает так, что для внедрения какой-либо задачи покупается новое оборудование. Поэтому аппаратное обеспечение нельзя обойти вниманием. Будем считать, что подбор подходящего оборудования также относится к оптимизации производительности InterBase.

Сервер для InterBase

Рассмотрим компоненты сервера согласно традиционным описаниям конфигурации компьютеров: платформа, процессор, материнская плата, ОЗУ, жесткие диски, сетевую плату.

Платформа. Под платформой понимается архитектура процессора, например Intel. Большинство серверов в мире продается на платформе Intel. Это неплохой и достаточно экономичный выбор. Существует версия клона InterBase 6 – Firebird – под платформу SPARC. Однако большинство заказчиков систем на базе InterBase вполне обойдутся серверами на базе платформы Intel, поэтому все дальнейшие рассуждения мы будем приводить, ориентируясь на Intel.

Процессор и материнская плата. Конечно, процессор важнейшая часть любой системы, когда дело касается интенсивных вычислений, однако рассматривать его в отрыве от материнской платы неразумно: даже самый лучший процессор, "посаженный" на низкопроизводительную материнскую плату, покажет довольно "средние" результаты. Сейчас на рынке конкурируют два сочетания "процессор и материнская плата" – от AMD и от Intel. Надо сказать, что по результатам тестов эти два конкурента практически сравнялись. Поэтому рекомендации будут лишь самого общего характера: процессор должен иметь большой внутренний кеш, а материнская плата – высокую частоту шины. Следует

воздержаться от дешевых процессоров на базе Celeron и Duron, а также от материнских плат с "урезанными" частотами шин (133 МГц и менее). Надо сказать, что частота шины дает серверу весьма значительное преимущество. Так, например, сервер, оснащенный процессором Intel Xeon 866 МГц, но с материнской платой, у которой частота шины 66 МГц, значительно проигрывает в производительности "простому" Pentuim 3–866, но "посаженному" на материнскую плату с частотой 133 МГц. Поэтому при выборе материнской платы для сервера следует обратить внимание на платы с частотой шины 266 МГц (для AMD-процессоров) и 400 МГц (для Intel-процессоров). Разумеется, закон Мура еще никто не отменял, и к моменту выхода книги ситуация на рынке процессоров может сильно измениться, но рекомендации – "высокая частота шины у материнской платы и большой кеш у процессора" – останутся прежними.

Два процессора и более. Несколько процессоров не дадут большого прироста производительности для InterBase, имеющего архитектуру SuperServer, но для архитектуры Classic позволят "почувствовать разницу" (подробнее о преимуществах и достоинствах различных вариантов InterBase см. ниже главу "Classic и SuperServer"). Поэтому если сервер выделен только под InterBase, а задача не требует использования Classic-архитектуры, то лучше сэкономить на многочисленных процессорах (и материнских платах для них) и выбрать однопроцессорный вариант.

ОЗУ. Оперативная память – весьма важный компонент серверной системы. Минимум ОЗУ, который позволит InterBase 6 работать, – это 32 Мбайт (возможно, и меньше; к сожалению, не нашлось компьютера с количеством ОЗУ 8 или 16 тМбайт, чтобы проверить). В наш век дешевых гигабайтов смешно говорить о таком небольшом количестве памяти, но InterBase действительно очень экономичен в использовании ОЗУ. Обычно количество ОЗУ рассчитывается следующим образом: необходимо взять среднее количество клиентов, одновременно использующих базу данных, выделить каждому по 10–15 Мбайт, затем прибавить 150 Мбайт. Например, если у вас 20 клиентов, то получим: 15*20+150 = 450 Мбайт. Нужно отметить, что величина в 10–15 Мбайт на клиента нужна в случае, если речь идет про архитектуру Classic, а клиенты весьма интенсивно работают с базой данных, например осуществляют аналитические выборки или что-то в этом роде. Вариант InterBase с архитектурой SuperServer значительно экономичнее расходует память – ему необходимо примерно на 30–40% меньше ОЗУ.

Если обратиться к реальной практике, то 50–60 клиентов системы АСТПП среднего класса (при размере базы данных около 1 Гбайт) неплохо обслуживаются InterBase SuperServer с 512 МБайт ОЗУ. Конкретные рекомендации таковы: для сервера, на котором ведется разработка, необходимо около 256 Мбайт ОЗУ, для рабочего сервера, обслуживающего задачу и среднего класса (т. е. 10 одновременно работающих активных пользователей), – 512 Мбайт и более. Количество ОЗУ также зависит от используемой ОС – для Linix обычно нужно меньшее количество ОЗУ для поддержания определенного уровня производительности, чем для Windows NT/2000.

Жесткие диски. Дисковая подсистема – один из наиболее важных компонентов сервера, который способен при неправильном его выборе чрезвычайно ухудшить производительность. Для InterBase-сервера неплохо использовать от-

казоустойчивые дисковые подсистемы на базе RAID5-массивов, причем желательно иметь контроллер с кеш-памятью не менее 32 Мбайт. Но это достаточно дорогое решение, поэтому для систем, не предъявляющих особых требований к производительности, можно остановиться на более дешевых IDE-накопителях, поддерживающих различные UDMA-режимы доступа. Крайне нежелательно использовать программный RAID-массив, эмулирующий зеркальное отображение информации, – это верный способ замедлить сервер.

Сетевая плата. Сервер должен обеспечивать надежное сетевое соединение с десятками и сотнями пользователей. Нельзя сказать, что дорогие сетевые платы, рекламируемые производителями как "специальные серверные решения", будут работать заметно быстрее обычных сетевых плат, установленных на клиентах. Сложно утверждать конкретно, в чем они надежнее обычных сетевых плат. В этой области рынка аппаратного обеспечения слишком много рекламного тумана, в котором спрятаны реальные преимущества серверных сетевых плат. Собственно, почти во всех случаях достаточно будет поставить на сервер, выделенный под InterBase, обыкновенную качественную сетевую плату – возможно, ту же, которая применяется на клиентах.

Сетевое оборудование

В настоящее время стандартом для корпоративных и офисных сетей является 100-Мбит Ethernet-сеть, основанная на витой паре (100Base-T). Она обеспечивает пропускную способность 3–10 Мбайт/с. 100-Мбит сеть достаточна для большинства клиент/серверных-приложений, в том числе и для приложений, использующих InterBase.

Обычно для организации сетевого общения InterBase и клиентов используется протокол TCP/IP, который является наиболее универсальным и масштабируемым сетевым протоколом. Практически все примеры в книге основываются на применении протокола TCP, хотя InterBase позволяет связываться и по протоколам Named Pipes и IPX.

Надо сказать, что неустойчиво работающая сеть, в которой часто теряются пакеты, является причиной возникновения ошибок InterBase 10054, загромождающих файл протокола InterBase.log. Поэтому, хотя вопросы настройки и обслуживания сети выходят за рамки данной книги, необходимо обратить особое внимание на обеспечение работы сети.

Рабочие станции

Требования к компьютерам-рабочим станциям, на которых исполняются клиентские части приложений базы данных на базе InterBase, определяются в основном требованиями ОС. Клиентская часть приложения базы данных InterBase не требует большего, чем обычные офисные программы. Клиентская же часть самого InterBase весьма легковесна, не требует много ресурсов и состоит всего из трех файлов (см. ниже главу "Состав модулей InterBase"). Можно с уверенностью сказать, что большинству приложений баз данных InterBase будет более чем достаточно для выполнения компьютера офисного класса – например, Celeron с 256 Мбайт ОЗУ. Нижним пределом (для небольшого клиентского приложения) является компьютер класса Pentium-100 с 16 Мбайт ОЗУ.

Основные "рычаги" управления производительностью

Сначала мы рассмотрим те параметры настройки InterBase, которые часто изменяют для настройки производительности InterBase. Рекомендации по улучшению производительности не только затрагивают те параметры, которые есть в конфигурационном файле ibconfig, но также рассматривают различные аспекты разработки программного обеспечения. Желающие разобраться во всех параметрах конфигурационного файла ibconfig могут обратиться к справочному материалу в приложении "Параметры конфигурационного файла InterBase".

Кеш базы данных

Кеш базы данных служит для хранения наиболее часто используемых страниц из базы данных. Его размер исчисляется в страницах и может быть установлен тремя разными способами:

- Заданием параметра файла конфигурации ibconfig DATABASE CASHE PAGES. При этом устанавливается значение кеша для каждой базы данных, обслуживаемой данным сервером; это означает, что для каждой подключенной базы данных будет распределен кеш указанного в данном параметре размера. Это не слишком гибкий способ, поэтому им обычно не пользуются, если компьютер-сервер обслуживает более одной базы данных. Этот параметр имеет следующие значения по умолчанию: InterBase 6 Classic (Linux) – 75, InterBase 6 SuperServer (Windows) – 2048, а InterBase 5.x – 256. Причем надо отметить, что для Classic устанавливается размер кеша для КАЖДОГО клиентского соединения.
- Установкой количества страниц для каждой индивидуальной базы данных с помощью инструмента gfix. Например, выполнение команды gfix –buffers 8192 для какой-либо базы данных приводит к тому, что все соединения к этой базе данных будут использовать кеш размером 8192 страницы. Помните, что версия Classic использует значение размера кеша для каждого клиентского соединения; поэтому не стоит устанавливать размер кеша в Classic более 2048 страниц (даже если у вас несколько гигабайтов ОЗУ). Необходимо отметить, что при создании базы данных задать размер кеша на уровне базы данных невозможно, – он устанавливается позже с помощью gfix.
- Установкой параметра функции InterBase API isc_dpb_num_buffers при соединении с базой данных.

Перечисленные способы установки идут в порядке возрастания приоритета, т. е. значения последних вариантов на практике надо использоваться в первую очередь, во всяком случае так указано в [4, гл. 6]. Тем не менее известный разработчик InterBase Ivan Prenosil [8] вносит поправки в это утверждение, считая, что установка кеша на уровне базы данных имеет наивысший приоритет и не может быть переопределена даже на уровне соединения.

Теперь о конкретных цифрах. Размер кеша может принимать значения от 50 до 65535 страниц. Чтобы посчитать, как много оперативной памяти потребует кеш базы данных, надо умножить размер страницы базы данных на размер кеша. Например, если страница имеет размер 8192 байта, а кеш имеет размер 5000

страниц, то потребуется 40960000 байт ОЗУ (около 40 Мбайт), чтобы вместить этот кеш. Несмотря на то что сейчас ОЗУ стоит достаточно дешево и экономить на нем не стоит, устанавливать размер кеша базы данных более 10000 страниц не следует, это вытекает из тестов, которые показывают уменьшение производительности InterBase при большом размере кеша. Помимо тестов, с результатами которых вы можете познакомиться на сайте www.ibase.ru, Ivan Prenosil указывает на малоизвестный факт: большой размер кеша заметно увеличивает время соединения с базой данной, и чем больше кеш, тем больше времени затрачивается на установку соединения. В последних версиях InterBase 7.1 SP2 и Firebird 1.5 эта проблема решена.

Forced Writes

Forced Writes – это режим записи данных на диск. Существует два режима – синхронный и асинхронный, которым соответствуют значения Forced Writes ON и OFF. Реально FW – это всего лишь флаг в атрибутах файла, показывающий операционной системе, как надо работать с данным файлом.

При асинхронном режиме записи данных на диск (т. е. при FW OFF) данные пишутся в файловый кеш ОС, в результате чего ускоряются операции с кешированными данными.

Очевидно, на разных операционных системах работа с файлом с включенным флагом асинхронной записи реализуется по-разному – Linux справляется очень хорошо, однако с Windows возникает опасность потерять очень много данных из-за сбоя сервера из-за сверхизбыточного кэширования файла базы данных.

Чтобы понять суть проблемы с кэшированием, необходимо знать когда производятся обращения к диску:

- когда происходит подтверждение транзакции: все страницы, затронутые изменениями в рамках этой транзакции, пишутся на диск;
- когда измененная страница становится "непопулярной": т. е. когда к ней перестают часто обращаться и нет смысла держать ее в кеше, она пишется на диск, а ее место в кеше занимает другая страница, к которой чаще обращаются;
- когда вытесняется несколько зависящих друг от друга страниц: запись одной из них на диск приводит к записи других;
- при использовании сервера InterBase с архитектурой Classic страница вытесняется из кеша одного пользователя и пишется на диск, если ее требует другой пользователь.

При включенном режиме асинхронного чтения данные пишутся на диск, когда этого пожелает ОС. При большом объеме ОЗУ база данных может почти целиком быть "втянута" в кеш. Тем не менее отключенный режим Forced Writes – это обоюдоострое оружие. Ускоряя операции чтения и записи данных, асинхронное чтение может привести к значительной потере данных в результате сбоя аппаратного или программного обеспечения. Кешированные данные могут находиться в кеше операционной системы длительное время (часы и дни), поэтому сбой электропитания может привести к потере данных, являющихся результатом часов и даже дней работы.

Поэтому стоит неоднократно подумать, прежде чем приносить надежность в жертву скорости и отключать Forced Writes. И конечно, не стоит включать режим асинхронной записи, если ваш сервер не оснащен источником бесперебойного питания.

Sweep Interval

Если посмотреть статистику по базе данных (как это сделать – см. в этой части главу "Статистика в InterBase"), то можно обнаружить в данных о заголовочной странице параметр Sweep interval. Этот параметр устанавливает разницу между старейшей интересующейся транзакцией (OIT) и следующей транзакцией (Next), при которой следует запускать процесс сборки "мусора". Подробнее о транзакциях и сборке "мусора" вы можете прочитать в главе, посвященной транзакциям (ч. 1), а с точки зрения производительности Sweep Interval интересует нас, поскольку при сборке "мусора" работа обычных клиентов, работающих с базой данных, может замедлиться. Особенно это актуально для серверов, работающих в круглосуточном режиме. Чтобы избежать проблем с производительностью, связанных с периодической сборкой "мусора", устанавливают Sweep Interval равным нулю, что означает запрет автоматической сборки "мусора". В этом случае процесс сборки "мусора" осуществляют вручную – например, с помощью резервного копирования. Дело в том, что процесс резервного копирования базы данных обычно сопровождается сборкой "мусора" (если не установлен флаг –garbage_collect). Полной заменой sweep процесс backup нельзя назвать, так как во время sweeping происходит обновление статуса транзакций. Поэтому если и отказываться от sweep, то необходимо заменить его не просто резервным копированием, а регулярным циклом backup/restore.

Чтобы установить sweep interval, используют инструмент gfix. Например, чтобы установить sweep interval в 0 и запретить автоматическую сборку "мусора", надо выполнить следующую команду:

```
gfix –h 0 –user SYSDBA –password <пароль>
C:\database\myDatabase.gdb
```

Размер страницы базы данных

На производительность операций чтения и записи сильно влияет размер страницы базы данных. Если страница мала (менее 4096 байт), то серверу приходится многократно обращаться к диску, чтобы прочитать некоторый кусок данных, т. к. все операции производятся постранично. Если страница имеет большой размер (8192 или даже 16384 байта), то нужно меньше "дергать" диск для чтения и записи данных. Помимо этих очевидных соображений, от размера страницы также зависит глубина индексов базы данных: чем страница больше, тем глубина меньше и тем быстрее происходит поиск с использованием индексов (подробнее см. главу "Индексы" (ч. 1)).

Для достижения оптимальной производительности рекомендуется устанавливать размер страницы базы данных не менее 4096 байт. Конечно, как всякое средство увеличение размера страницы имеет и побочный эффект – приводит к росту размера файлов базы данных.

Установить или изменить размер страницы базы данных можно только при ее создании или при восстановлении из резервной копии (см. главу этой части). "Резервное копирование и восстановление из резервной копии".

Интересно ознакомиться с общими рекомендации по оптимизации от известного разработчика Далтона Калфорда, которыми он поделился в одном из писем в конференцию mers.com, посвященную вопросам использования InterBase. Они включают следующие пункты:

1. Разнесите файлы ОС, каталог временных файлов InterBase и файлы базы данных на разные каналы (и на разные диски, соответственно). Это означает, что ОС и программы (в том числе и InterBase) нужно разместить на одном диске временные файлы ОС и InterBase – на другом диске, а файлы базы данных поместить на надежном RAID.
2. Разнесите медленные и быстрые SCSI-устройства на разные каналы, т. к. многие SCSI-адаптеры работают на скорости самого медленного устройства. Поэтому следите за тем, чтобы CDROM, сканеры или ленточные накопители не были подключены к основному каналу, на котором расположены SCSI-диски с вашими базами данных.
3. Так как InterBase может использовать многофайловые базы данных, расположенные на разных дисках/контроллерах/RAIDax, то в случае использования нескольких RAID-контроллеров лучше всего расположить разные части базы данных на разных дисках, в результате чего каждый контроллер будет обслуживать свою порцию базы данных.
4. InterBase создает временные файлы на диске во время выполнения любых запросов, которые могут дать в качестве результата неопределенное количество записей (а это практически все SQL-запросы на выборку данных). Кеш InterBase предназначен лишь для хранения страниц из базы данных, а не для кеширования временных файлов. Учитывая, что большинство ОС слабо оптимизированы для работы с временными файлами, для хранения временных файлов InterBase лучше всего создать диск в памяти (RAM-диск), а в файле конфигурации ibconfig с помощью параметров TMP_DIRECTORY установить первый каталог для хранения временных файлов на этот RAM-диск. Это должно заметно улучшить производительность запросов на выборку данных.
5. Не разрешайте совместное использование дисков, на которых находятся базы данных. Чтение и запись данных из базы данных достаточно нагружают дисковую подсистему. Если разрешать использовать тот же диск как хранилище пользовательских файлов, то производительность может заметно ухудшиться.
6. Используйте для организации сети как можно меньше протоколов. Лучший выбор – это применять TCP/IP. Множество протоколов может пересекаться и создавать неприятные коллизии, которые могут значительно ухудшить пропускную способность сети.
7. Выделите специальный сервер для InterBase-сервера, т. е. не делайте этот компьютер одновременно своим Web-сервером, почтовым сервером и сервером приложений. Помимо загрузки компьютера, дополнительные приложения могут в результате своих ошибок повредить ОС, переполнить диск или еще каким-то образом осложнить вам жизнь.

Заключение

Необходимо уделять особое внимание оптимизации производительности приложений баз данных InterBase, так как правильно выбранные настройки позволяют улучшать производительность без чрезмерных затрат средств на покупку аппаратного обеспечения и эксплуатировать приложения баз данных с максимальной эффективностью.

Безопасность в InterBase: пользователи, роли и права

Особенности системы защиты данных в InterBase

Легкость и доступность информации, которые принесли с собой компьютерные технологии, имеют и свою обратную сторону – использование компьютеров резко обострило проблемы сохранности и конфиденциальности данных. Информация, которая хранится в базе данных, зачастую может стоить во много раз дороже иного бриллианта или золотого слитка.

Поэтому обеспечение безопасности хранимых данных является неотъемлемой частью любой современной СУБД, в том числе и InterBase.

InterBase предоставляет развитые средства для управления безопасностью в своих базах данных. Но, прежде чем рассказать о конкретных способах защиты данных, необходимо прояснить ряд моментов в концепции безопасности InterBase, которые часто смущают пользователей, знакомых с другими СУБД.

Как и в большинстве других СУБД, в InterBase защита данных основывается на том, что существует концепция *пользователей*, которые получают те или иные *права* для работы с каждым объектом внутри базы данных. Реальные люди-пользователи получают в свое распоряжение имя пользователя InterBase и его пароль и применяют его для работы с базой данных.

Под пользователем InterBase мы будем понимать регистрационную запись, состоящую из имени пользователя и идентифицирующего его пароля.

Администратор СУБД InterBase заводит необходимое число пользователей и назначает им нужные для их работы права, разрешая им доступ только к той информации, которая нужна для выполнения должностных обязанностей.

В этом InterBase как две капли воды похож практически на любую СУБД. Однако есть существенное отличие – данные о пользователях базы данных хранятся не в самой базе данных, а вне ее – в особой базе данных пользователей *ISC4.gdb*.

Дело в том, что реализация ограничений, налагаемых на объекты базы данных, осуществлена в InterBase на уровне сервера базы данных, а не самой базы данных. Это означает, что внутри базы данных данные никак не шифруются и не защищаются. Все проверки прав доступа осуществляются сервером InterBase, который сравнивает права, выданные на объект базы данных, с правами, которые имеет конкретный пользователь, и в зависимости от результатов сравнения, разрешает или не разрешает доступ этого пользователя к запрашиваемому объекту.

Следствием вынесения информации о пользователях и проверки прав доступа к базе данных на уровень сервера является то, что, физически скопировав базу данных на другой сервер, мы можем воспользоваться паролем администратора этого сервера и получить полный доступ к информации в базе данных, обойдя, таким образом, все ограничения на доступ к данным.

Причиной такого решения является, по-видимому, особый взгляд компании Borland на физическую безопасность файлов баз данных. Считается, что защиту *файла* базы данных необходимо обеспечивать на ином уровне, чем уровень СУБД. Прежде всего на уровне ОС – путем запрета сетевого доступа к файлам базы данных и установки соответствующих прав доступа на каталоги и файлы

баз данных (ниже мы подробно рассмотрим рекомендации по установке прав доступа на файлы базы данных). Затем следует ограничить доступ к компьютеру-серверу, на котором хранится база данных, чтобы предотвратить физический доступ злоумышленника к носителям, на которых хранятся файлы базы данных (а также их резервные копии).

Другими словами, InterBase обеспечивает управление безопасностью лишь в рамках своей компетенции, т. е. его система защиты информации предназначена исключительно для ограничения доступа к данным пользователей InterBase. Заботиться о том, чтобы только этот способ доступа к базе данных стал единственным, – задача не InterBase, а опытного системного администратора и службы безопасности предприятия.

Размещение пользователей InterBase в отдельной базе данных позволяет во всех базах данных, обслуживаемых данным сервером, использовать единое пространство имен пользователей, что может упростить настройку и администрирование системы безопасности.

Разрушаем легенду

Несколько лет назад по всему IT-миру прокатился слух, что внутрь кода InterBase встроен универсальный пользователь и пароль, позволяющий получить доступ к любой базе данных под управлением InterBase. Имя пользователя было politically, а пароль – corrected. Надо сказать, что такой слух имел под собой основания – в одной из промежуточных версий InterBase 5.x действительно была такая "дыра" в безопасности. Однако компания Borland практически мгновенно отреагировала и выпустила "заплатку" для ликвидации этой проблемы. С тех пор в InterBase нет подобных проблем – после того, как были обнародованы исходные коды 6-й версии, в этом может убедиться каждый желающий.

Хочется отметить, что в сочетании politically/corrected (политически корректный) кроется своеобразная ирония. Как известно, принципы "политической корректности" провозглашают отсутствие двойных стандартов. Всем нам известно множество примеров, когда IT-компании "первого эшелона" допускали появление в своих продуктах (в том числе и в СУБД) многочисленных "дыр", приводивших к многомиллионным потерям. Однако про эти проблемы никто не вспоминает, а про встроенного пользователя в InterBase, который, кстати говоря, имел весьма ограниченные права, никак не могут забыть. Впрочем, может, не вспоминают эти проблемы потому, что они активно вытесняются новыми, более "свежими" прорехами в безопасности, а про InterBase, кроме этого случая, вспоминать нечего.

Как бы то ни было, вы можете сами проверить уязвимость InterBase. Попробуйте подключиться к базе данных как politically/corrected и получить несанкционированный доступ к базе данных!

Система безопасности InterBase

Разъяснив ряд "идеологических" особенностей защиты информации в InterBase, можем перейти к конкретному описанию системы безопасности этой СУБД и ее применению на практике. Начнем наше рассмотрение с основных понятий, которыми оперирует система безопасности.

Пользователи

Пользователь InterBase, как уже было сказано выше – это регистрационная запись, доступная во всех базах данных, обслуживаемых данным сервером. Пользователи InterBase, как правило, хранятся в служебной базе данных ISC4.gdb, но если и клиент, и сервер InterBase стоят на системе Linux/Unix, т. е. еще одна возможность. InterBase распознает пользователей и даже группы пользователей ОС Unix и поэтому можно рассматривать Unix-пользователей как обычных пользователей InterBase, несмотря на то, что они не занесены в служебную базу данных InterBase ISC4.gdb. Для осуществления такой возможности используется механизм "доверенных компьютеров" (trusted hosts). Пользователи Windows-клиентов (и тем более Windows-серверов InterBase) лишены такой возможности.

Среди всех пользователей главнейшим, несомненно, является пользователь SYSDBA – системный администратор сервера InterBase. Имя SYSDBA предопределено и не может меняться. По умолчанию этот пользователь обладает всеми правами над любым объектом базы данных. Поэтому SYSDBA является очень мощным пользователем и следует тщательно оберегать его пароль. Как вы знаете из предыдущих глав, по умолчанию пароль системного администратора – "masterkey" (на самом деле в InterBase используется всего 8 символов, и достаточно писать "masterkey"), однако желательно сменить этот пароль сразу после установки сервера и регулярно менять его впоследствии.

Чтобы создать нового пользователя, необходимо воспользоваться либо инструментом командной строки gsec, либо каким-либо графическим инструментом из списка, приведенного в приложении "Инструменты администратора и разработчика InterBase". С помощью SQL-команды создать или удалить пользователя InterBase нельзя – это следствие вынесения системы безопасности на уровень сервера.

Имя пользователя может иметь длину до 31 символа включительно, пароль – до 32 символов, однако из них для аутентификации используются только первые 8.

> Для имен пользователей не важен регистр символов, но пароль является регистрочувствительным.

Надо отметить, что только SYSDBA может создавать и изменять новых пользователей в InterBase. Для создания с помощью gsec нового пользователя с именем TESTUSER необходимо выполнить следующую команду:

```
gsec -user SYSDBA -password masterkey -add TESTUSER -pw test
```

Чтобы просмотреть с помощью gsec список пользователей InterBase, необходимо воспользоваться ключом –display:

```
C:\Firebird\bin>gsec -display
     user name                       uid   gid     full name
------------------------------------------------------------
SYSDBA                               0     0
BUILDER                              0     0
TESTUSER                             0     0
```

Подробно ключи и использование инструмента командной строки gsec описаны в [4, гл. 5]. Мы не будем здесь повторять приведенный материал, так как основные команды для управления совершенно очевидны, однако вы всегда сможете познакомиться с ними поближе.

Обычно в системе на базе InterBase заводят столько пользователей, сколько людей обращается к приложению базы данных. Если же имеется несколько человек, выполняющих одну и ту же работу, то для упрощения системы прав применяют механизм *ролей* (ROLE).

Роли

Роли InterBase – это своего рода суррогатные пользователи. Роли служат для организации пользователей с одинаковыми правами в группы. Например, если у нас имеется группа пользователей, для которых нужен доступ только на чтение, то мы создаем роль с именем READER, присваиваем этой роли все необходимые права и затем можем назначать эту роль со всеми принадлежащими ей правами какому-то конкретному пользователю.

Вы можете поинтересоваться, зачем нужно вводить промежуточный механизм ролей, когда права можно напрямую давать пользователям InterBase, соответствующим конкретным людям. Во-первых, люди приходят и уходят, а исполняемые ими роли остаются. Во-вторых, если давать права напрямую пользователям, а не посредством ролей, то при изменении прав у определенной группы пользователей придется модифицировать права у каждого пользователя. А при использовании роли достаточно изменить права только у роли, чтобы у всех принадлежащих ей пользователей изменились права. В-третьих, большое количество прав для разных пользователей на все объекты базы данных может значительно увеличить объем данных системной таблицы RDB$USER_PRIVILEGES. Это может сказаться на скорости выполнения всех запросов, поскольку для любого запроса InterBase проверяет наличие соответствующих прав у пользователя, который выполнил запрос.

Роли – это объекты уровня базы данных. Они видны только внутри той базы данных, в которой определены. Фактически роли представляют собой записи в системной таблице RDB$ROLES. Чтобы создать роль с именем READER, необходимо выполнить следующий DDL-запрос (его можно выполнить в isql или в каком-либо графическом инструменте):

```
CREATE ROLE READER;
```

При использовании механизма ролей при соединении с базой данных надо указывать и имя пользователя, и его желаемую роль. Обычно роль указывается в параметрах соединения с базой данных.

Теперь мы вплотную подошли к третьему ключевому понятию системы безопасности InterBase – к правам. Именно права являются тем, что наполняет конкретным смыслом понятия "пользователь" и "роль".

Права

Права в InterBase – это разрешение какому-либо пользователю, хранимой процедуре или триггеру совершить какую-либо операцию над определенным объектом базы данных. Существуют несколько видов объектов, на которые

можно устанавливать права для пользователей и ролей. Это таблицы и их поля, представления и хранимые процедуры.

Существуют следующие права, выдаваемые пользователям InterBase:

1. Для таблиц и их полей – права на выполнение операций SELECT, DELETE, INSERT, UPDATE и REFERENCES (это право дает пользователю возможность создавать ограничения внешнего ключа FOREIGN KEY на данную таблицу).
2. Для представлений (VIEW) и их полей – права на выполнение операций SELECT, DELETE, INSERT, UPDATE. Как вы уже знаете из главы "Представления" (ч. 1), данные представления не хранятся в базе данных, а извлекаются динамически по запросу, лежащему в основе представления. Поэтому права на изменение данных в представлении (DELETE, INSERT, UPDATE) фактически относятся либо к таблице, на которой основано представление, либо к триггерам, которые делают VIEW изменяемым.
3. Права на выполнение хранимых процедур – EXECUTE. Пользователь, желающий выполнять определенную хранимую процедуру, должен иметь на это право.

Помимо прав на объекты, которые выдаются пользователю InterBase, права на использование объектов могут иметь также те объекты, которые на них ссылаются. Например, если хранимая процедура производит вставки в какую-либо таблицу, то этой процедуре должны быть даны права на INSERT в этой таблице.

Вы могли также заметить, что часто создавали хранимые процедуры и триггеры, не заботясь о выдаче каких-либо прав, и все они прекрасно работали. Действительно, можно создать хранимую процедуру и она будет вставлять данные в таблицу, даже если она не имеет непосредственного разрешения на вставку – но только в том случае, если выполняющий ее пользователь InterBase имеет эти права! Другими словами, хранимая процедура применяет список прав выполняющего ее пользователя для своей работы.

Такой подход является чрезвычайно гибким и прозрачным. Например, мы можем вообще не заботиться о раздаче прав на хранимые процедуры и всю безопасность определять на уровне прав, выдаваемых пользователю. Это удобно в случае, если мы работаем в нашем приложении с таблицами базы данных напрямую: выполняем какие-либо вставки, изменения и удаления.

Однако также можно вынести всю логику работы с данными в хранимые процедуры, раздать им необходимые права на таблицы, а всем пользователям InterBase полностью запретить доступ к таблицам базы данных, чтобы исключить прямую правку данных, и разрешить только применение хранимых процедур. В этом случае все пользователи InterBase будут изолированы от непосредственных данных в таблицах базы данных и смогут работать с ними лишь посредством хранимых процедур, в которых легко организовать отслеживание изменений, и т. д.

Раздача прав

Права на объекты базы данных раздаются с помощью команды GRANT. Это очень многоликая команда, со множеством опций, поэтому мы не будем целиком цитировать документацию, а приведем лишь самые основные и часто используемые применения этой команды.

Давайте рассмотрим несколько простых примеров раздачи прав. Чтобы выдать пользователю TESTUSER права на выборку из таблицы Table_Example, применяется следующая команда:

```
GRANT Select ON Table_Example TO testuser;
```

Как видите, все очень просто и интуитивно понятно. Чтобы выдать права на чтение и запись данных, но не на изменение, используем следующую команду:

```
GRANT Select, Insert ON Table_Example TO testuser;
```

А чтобы выдать все возможные права, применяется ключевое слово ALL:

```
GRANT ALL ON Table_Example TO testuser;
```

Часто возникает необходимость выдать определенные права сразу нескольким пользователям. Для этого можно указать в команде GRANT не одного, а сразу нескольких пользователей (до 1500) через запятую следующим образом:

```
GRANT ALL ON Table_Example TO testuser, newuser;
```

Если же надо выдать определенные права сразу всем пользователям InterBase, то можно воспользоваться ключевым словом PUBLIC, которое эквивалентно перечислению всех пользователей через запятую:

```
GRANT Select, Insert ON Table_Example TO PUBLIC;
```

Помимо выдачи прав на всю таблицу (или представление – синтаксис здесь будет точно таким же), иногда возникает необходимость ограничить права пользователя несколькими определенными полями в таблице. Например, пользователь BOSS имеет право менять поле BIGMONEY, а все остальные пользователи могут только его просматривать. В этом случае можно воспользоваться механизмом ограничения выдачи прав на конкретные поля таблицы:

```
GRANT Select ON Table_example(BIGMONEY) TO PUBLIC;
GRANT ALL ON Table_example(BIGMONEY) TO BOSS;
```

Аналогичный синтаксис раздачи прав применяется и для хранимых процедур, только в первой части команды GRANT пишется GRANT EXECUTE ON PROCEDURE <имя_процедуры>, а далее как обычно. Например, для выдачи пользователю TESTUSER разрешения запускать хранимую процедуру SP_ADD надо написать следующее:

```
GRANT EXECUTE ON PROCEDURE SP_Add TO testuser;
```

Для того чтобы некая процедура SP_MAIN могла вызывать процедуру SP_ADD без наличия таких прав у пользователя, вызывающего SP_MAIN, надо написать так:

```
GRANT EXECUTE ON PROCEDURE SP_Add TO SP_Main;
```

Однако раздача прав объектов на использование других объектов нужна только в том случае, если база данных проектируется с намерением скрыть исходные таблицы от пользователей. В противном случае гораздо проще будет раздавать права напрямую пользователям (ролям).

Организация пользователей в группы с помощью ролей

Чтобы уменьшить количество выдаваемых разрешений и объединить пользователей в группу по принципу наличия у них одинаковых прав, применяется механизм ролей. Порядок действия для использования ролей следующий:

1. Необходимо создать роль.
2. Выдать этой роли достаточные права.
3. Назначить конкретным пользователям эту роль.
4. При подсоединении к базе данных указать, помимо имени пользователя, ту роль, права которой будет иметь в течение времени данного соединения пользователь. То, что роль явно указывается при подсоединении, позволяет иметь для каждого пользователя набор ролей и менять их при каждом сеансе в зависимости от характера выполняемой работы.

Приведем пример использования ролей. Для начала создадим роль с именем READER, которая будет иметь права для чтения данных:

```
CREATE ROLE READER;
```

Выдадим этой роли права на чтение таблицы `Table_example`:

```
GRANT Select ON Table_example TO READER;
```

Присвоим эту роль пользователю TESTUSER, чтобы он мог указать ее при подсоединении к базе данных (если этого не сделать, то возникнет ошибка авторизации):

```
GRANT READER TO testuser;
```

Теперь мы можем указать эту роль при подсоединении к базе данных. Все современные библиотеки доступа, описанные в данной книге, имеют специальную опцию для использования роли пользователя в течение соединения. За подробностями обращайтесь к соответствующим главам. В общем виде, например при подсоединении через isql, указание роли выглядит так:

```
CONNECT 'server:Disk\Path\database.gdb' USER 'username'
PASSWORD 'password' ROLE ' rolename';
```

Для нашего примера и тестовой базы данных firstbase.gdb строка соединения по TCP/IP будет выглядеть следующим образом:

```
CONNECT 'localhost:C:\Database\firstbase.gdb' USER 'sysdba'
PASSWORD 'masterkey' ROLE 'READER';
```

Использование ролей, которое возможно в InterBase начиная с версии 5.x, позволяет значительно упростить управление правами пользователей InterBase.

Аннулирование прав

Совершенно очевидно, что поскольку права могут быть выданы, то их можно и отобрать. Для этого существует команда REVOKE. В принципе она представляет собой копию GRANT, только с обратным действием. Формат команды REVOKE для различных объектов базы данных похож на GRANT. Например,

чтобы отобрать право чтения таблицы table_example у пользователя TESTUSER, достаточно написать:

```
REVOKE Select ON Table_example FROM testuser;
```

Точно так же, как и в GRANT, в REVOKE можно перечислять пользователей и права через запятую, применять "псевдонимы" ALL для удаления всех прав (вне зависимости от того, есть они или нет) и PUBLIC для аннулирования прав сразу у всех пользователей. С помощью REVOKE можно также лишить пользователя назначенной ему роли или аннулировать какие-то права у самой роли. Совершенно очевиден также тот факт, что невозможно как-то ограничить или расширить права пользователя SYSDBA. Если бы это было возможно, то в системе защиты InterBase содержалось бы явное противоречие: пользователь SYSDBA мог бы отобрать права на раздачу прав сам у себя, соответственно без права их восстановить! Таким образом, следует помнить, что пользователь SYSDBA всегда обладает всеми возможными правами.

Не будем утомлять читателя демонстрацией примеров употребление всех возможных применений REVOKE. При необходимости все эти вопросы можно прояснить с помощью документации [1, гл. 12]. Теперь мы перейдем к значительно более важному вопросу – к идеологии применения прав.

Как правильно раздавать и аннулировать права

Предыдущее разделы описывали практические примеры раздачи и аннулирования на объекты базы данных. Однако система безопасности – это всегда иерархическая система, в которой есть более ответственные пользователи, раздающие различные права менее ответственным.

Сейчас настало время прояснить схемы раздачи прав. Прежде всего необходимо ввести понятие *владельца объекта* (owner). Владелец объекта – это тот, кто создал его. Если пользователь TESTUSER создал какую-то таблицу, то он является владельцем этой таблицы.

Обычно все объекты в период разработки базы данных создаются одним пользователем – SYSDBA. С применением этого же пользователя, как правило, производится вся разработка клиентского приложения. В результате получается, что все объекты всегда доступны. Когда появляется необходимость ввести разграничения по пользователям, необходимо регулировать множество прав, причем не всегда можно заранее сказать, какие права и на какие объекты могут понадобиться для нормальной работы приложения. Из-за этого начинающие разработчики часто считают права на объекты "излишеством" и стараются придумать собственные системы безопасности, не утруждая себя изучением уже существующей. Если вы не хотите попасть в их число, то мы рекомендовали бы вам попытаться разобраться в этой ситуации.

Итак, по умолчанию права на любой объект в InterBase, будь то таблица, представление или хранимая процедура, имеет только его владелец, а также системный администратор SYSDBA. Соответственно раздавать права по умолчанию может только владелец объекта. Любой другой пользователь, не являющийся владельцем объекта, не сможет выдать другому пользователю права на этот объект, если только владелец объекта не передал другому пользователю соот-

ветствующие права со специальной опцией WITH GRANT OPTION. Указание этой опции в конце обычного предложения GRANT означает, что пользователь не только получает эти права, но и сможет передавать их другому пользователю. Например:

```
GRANT Select ON Table_example TO testuser WITH GRANT OPTION;
```

Теперь пользователь testuser может не только выбирать записи из таблицы `Table_example`, но также передавать право Select (и только его!) другим пользователям.

Если теперь пользователь testuser выдаст права пользователю newuser, затем владелец базы данных отберет право на SELECT у пользователя testuser, то автоматически newuser также потеряет права на Select. То есть все права, выданные пользователем testuser, будут аннулированы.

Для того чтобы не возникало проблем с правами, после этапа активного изменения метаданных лучше всего отказаться от использования SYSDBA как основного пользователя, а создать "специального" пользователя и применять его для разработки клиентского приложения.

Передача прав

Часто случается так, что во время разработки базы данных программист динамически добавляет необходимые права каким-либо пользователям, однако документировать вносимые изменения забывает. Для того чтобы выяснить права какого-либо пользователя, можно извлечь данные из системных таблиц InterBase. Для извлечения всех прав пользователя TESTUSER можно употребить следующий SQL-запрос:

```
SELECT    RDB$USER, RDB$GRANTOR, RDB$PRIVILEGE,
    RDB$GRANT_OPTION, RDB$RELATION_NAME, RDB$FIELD_NAME,
    RDB$USER_TYPE, RDB$OBJECT_TYPE
FROM  RDB$USER_PRIVILEGES
WHERE RDB$USER = 'TESTUSER'
```

В результате этого запроса получим таблицу, содержащую все права, выданные пользователю TESTUSER. Применяя механизмы поиска и замены в любом текстовом редакторе, можно легко превратить возвращаемую таблицу в полноценные команды GRANT, получив, таким образом, скрипт, который можно будет использовать для "раздачи прав".

Чтобы назначить пользователю NEWUSER такие же права, как и у пользователя TESTUSER, причем сделать это от имени SYSDBA, можно применить следующий SQL-запрос:

```
INSERT INTO RDB$USER_PRIVILEGES (RDB$USER, RDB$GRANTOR,
    RDB$PRIVILEGE, RDB$GRANT_OPTION, RDB$RELATION_NAME,
    RDB$FIELD_NAME, RDB$USER_TYPE, RDB$OBJECT_TYPE)
SELECT 'NEWUSER', 'SYSDBA', RDB$PRIVILEGE, RDB$GRANT_OPTION,
    RDB$RELATION_NAME, RDB$FIELD_NAME,
    RDB$USER_TYPE, RDB$OBJECT_TYPE
FROM RDB$USER_PRIVILEGES UPR
WHERE UPR. RDB$USER = 'TESTUSER'
```

Разумеется, такой подход к манипулированию является недокументированным и может измениться и не работать в дальнейших версиях InterBase и его клонов, но иногда его использование может сэкономить массу времени на кропотливое создание SQL-скриптов со множеством GRANT.

Особенности InterBase 6.5

В отличие от предыдущих версий, InterBase 6.5 имеет несколько другие права по умолчанию на системные таблицы. Пользователь SYSDBA, разумеется, имеет все права, однако все остальные могут только читать данные из системных таблиц. Это было сделано для того, чтобы избежать возможности прямой несанкционированной правки системных таблиц, что может привести к потере нужных метаданных. Разумеется, при использовании любых других версий InterBase вы можете достичь того же результата, применяя команду REVOKE ко всем системным таблицам. Также следует обратить внимание, что в InterBase 6.5 утилита gbak при восстановлении базы данных из резервной копии также восстанавливает заданные права на системные таблицы. Gbak из предыдущих версий InterBase восстанавливал права на все таблицы, кроме системных.

Общие рекомендации по безопасности

Чтобы избежать взлома вашей системы, необходимо соблюдать простые рекомендации, которые значительно осложнят жизнь злоумышленнику, если он вдруг попытается добраться до вашей информации.

1. Для установки и функционирования сервера InterBase используйте промышленные ОС, в которых предусмотрена система безопасности. Это Windows NT/2000/XP, Linux/Unix/Solaris. Избегайте устанавливать сервер с важной информацией на домашние ОС – Windows 95/98/Me. Конечно, для действительно защищенной системы необходимо наличие квалифицированного системного администратора, который должен контролировать все попытки доступа к компьютеру-серверу.
2. Используйте средства ОС для ограничения диапазона IP-адресов к компьютеру-серверу. Таким образом можно значительно осложнить атаку через Интернет.
3. Используйте нестандартный порт для работы с сервером InterBase. По умолчанию применяется порт 3050, но вы сможете изменить это значение на какое-то другое. Номер используемого порта настраивается в файле services. В этом случае в строке соединения с базой данных надо указать номер порта. Например, для порта 4671 строка соединения будет иметь вид srv:4671:Disk\Path\file.gdb. Такую возможность поддерживает все современные клоны InterBase 6.x: Firebird 1.x, Yaffil 1.0 и InterBase начиная с 6.5.
4. Не устанавливайте сервер InterBase по стандартному пути, предлагаемому установщиком, а также не используйте очевидные пути вроде "C:\IBServer". Это поможет осложнить хищение служебной базы данных ISC4.gdb, если злоумышленник получит удаленный доступ к серверу.

5. Не разрешайте совместное использование (shared access) по сети тех каталогов, которые содержат файлы баз данных. Это не имеет никакого смысла (см. раздел "Строка соединения" в главе "Создаем базу данных" (ч. 1)).
6. При работе на Windows NT/2000/XP используйте файловую систему NTFS. Установите разрешения файловой системы на файлы базы данных только для пользователя "SYSTEM" (или для того пользователя, под чьим именем выполняется серверный процесс InterBase – это можно узнать с помощью апплета "Службы" (Services) в панели управления Windows). Для подключения к базе данных InterBase по рекомендованному протоколу TCP/IP удаленный клиент базы данных должен иметь права только на операцию соединения с сервером. Другими словами, удаленный клиент не работает напрямую с файлами базы данных – с ними работает только сам серверный процесс InterBase. Однако если удаленный клиент работает с InterBase под Windows по протоколу NetBeui (проще говоря, если в приложении используется строка соединения вида \\srv\Disk\Path\file.gdb) и файл базы данных расположен на диске с файловой системой NTFS, то для работы с базой данных этому клиенту потребуются разрешения NTFS на этот файл. Это замечание имеет смысл только для сетей на базе Windows NT.

Что такое "архитектура сервера СУБД"?

Архитектура – понятие столь широкое и столь часто употребляемое, что, пожалуй, стоит определить, что мы будем понимать под архитектурой в нашем конкретном случае, т. е. по отношению к серверу баз данных InterBase. Попытаемся очертить круг проблем, интуитивно связываемых с понятием *архитектуры сервера СУБД*. Прежде всего, это способ хранения и обработки информации в базе данных. По этому принципу СУБД можно подразделить на реляционные, сетевые, объектно-ориентированные, иерархические и т. д. InterBase относится к реляционным СУБД, и останавливаться на том, что это такое, мы не будем. Коротко определения этих видов СУБД приведены в глоссарии в конце книги. Если читатель пожелает познакомиться с ними поближе, то лучше всего обратиться к какой-нибудь из множества превосходных книг по этому вопросу, доступных в печатном виде или в Интернете.

Второй важной стороной понятия архитектуры является способ взаимодействия клиентов – потребителей данных с сервером, которые эти данные хранит и обрабатывает. Обычно способ обмена каким-то образом именуется: "архитектура клиент-сервер", "многозвенная архитектура" или "локальные базы данных". Общего, объединяющего названия у этих способов обмена нет, но, несмотря на недостатки подобной классификации, можно сказать, что InterBase представляет собой систему клиент-серверной архитектуры. Под понятием "клиент-серверная архитектура" понимают массу различных вещей. Общим у всех систем, к которым можно применить определение "клиент-серверная", пожалуй, является тот факт, что такая система всегда имеет две четко разделенные части – клиентскую и серверную. В связи с таким делением часто возникает путаница с терминами "сервер" и "клиент". Давайте сразу внесем ясность в этот вопрос. Существуют следующие виды "серверов":

- Сервер как компьютер-сервер, т. е. отдельная ЭВМ, обслуживающая запросы, приходящие с других компьютеров.
- Сервер как экземпляр серверной части СУБД InterBase, выполняющий запросы клиентской части СУБД. Обратите внимание, что серверная и клиентская части СУБД InterBase не обязательно должны находиться на разных компьютерах – они могут выполняться и на одном.

Под понятием "клиент" можно понимать как компьютеры, на которых выполняются какие-то конкретные прикладные программы, так и сами эти программы, которые используют СУБД. Также под клиентом может пониматься клиентская часть InterBase, которая необходима для передачи запросов от прикладных программ серверной части СУБД.

Схематично архитектура клиент-сервер в ее типичной конфигурации изображена на рисунке 4.1.

В этой книге под "сервером" мы будем понимать серверную часть СУБД InterBase, а под "клиентом" – его клиентскую часть. Если эти термины будут использоваться в другом смысле, то это будет указано.

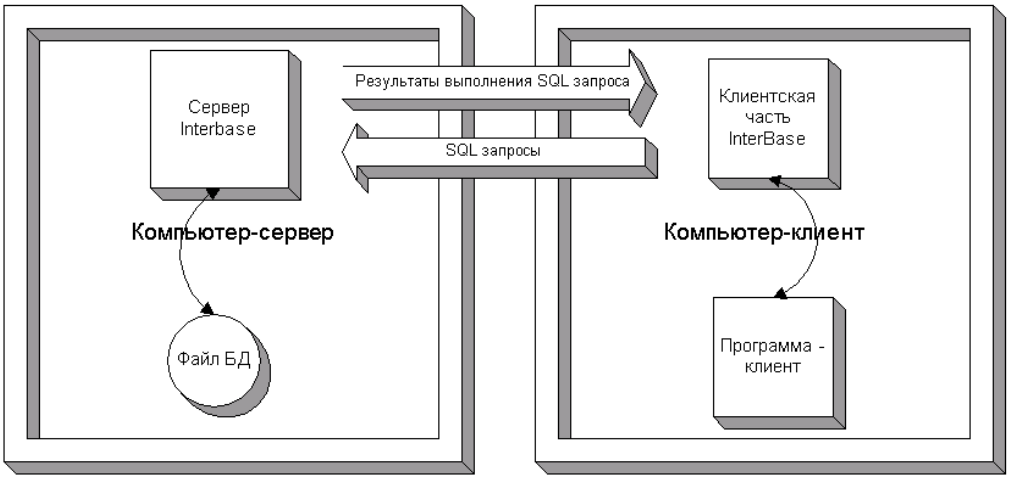


Рис. 4.1. Архитектура клиент-сервер InterBase

Под архитектурой также часто понимается состав программного комплекса, т. е. то, из каких модулей состоит сложная сущность, называемая обычно "сервером СУБД", как эти модули взаимодействуют и обеспечивают работу СУБД. InterBase имеет две различные реализации, которые имеют разную архитектуру взаимодействия модулей: Classic и SuperServer. Эти две различные архитектуры будут рассмотрены в главе "Classic и SuperServer". Там мы узнаем, чем отличаются эти архитектуры, и какие недостатки и преимущества они имеют.

И пожалуй, самое простое определение – это архитектура как "начинка" сервера, как ответ на "детский" вопрос, как это все устроено и почему оно работает так, а не иначе. Какова "начинка" СУБД, как организуются данные во все более сложные структуры, начиная от расположения байтов на диске и заканчивая логическими объектами, которые позволяют легко манипулировать данными в базе данных? Ответ на этот вопрос будет рассмотрен в главе "Структура базы данных InterBase". К рассмотрению "начинки" InterBase мы приступим с самого "низа" – с физической структуры данных, посмотрим, как на самом деле лежат байты и биты тех данных, которые пользователи вверяют InterBase. Затем мы перейдем к тому, как элементарные элементы данных организуются в более сложные структуры, которые отвечают логике прикладных задач, обычно решаемых программистами баз данных, т. е. к таблицам, триггерам, хранимым процедурам и другим объектам логической структуры базы данных.

Состав модулей InterBase

Давайте подробнее рассмотрим, из каких файлов состоит сервер и клиент InterBase. Эта информация нужна для того, чтобы понимать, из каких ключевых модулей состоит сервер, и уверенно осуществлять администрирование сервера, например смену версий сервера.

В этой главе мы рассмотрим вариант InterBase с архитектурой SuperServer (SS) под Windows, а также Classic Server (CS) под Linux. Состав модулей, входящих в InterBase на других ОС, рассматривать здесь не будем, поскольку отличия будут незначительные.

InterBase Super Server для Windows

Итак, что попадает на компьютер в результате установки сервера InterBase SuperServer под Windows? Чтобы это выяснить, необходимо изучить содержание установочного каталога InterBase. Таблица 4.16 коротко описывает назначение каталогов и файлов, входящих в состав InterBase SS.

Таблица 4.16. Каталоги и файлы в установочном каталоге IB SS for Windows

Каталог или файл	Краткое описание
Каталоги	
BIN	Содержит набор исполняемых файлов, которые реализуют все основные функции InterBase-сервера. Подробнее см. ниже в разделе "Каталог BIN для SuperServer"
HELP	В этом каталоге находится база данных help.gdb, которая содержит краткую справку по командам и ключевым словам InterBase SQL
INCLUDE	Содержит несколько заголовочных файлов на языке C, которые могут быть использованы разработчиками, желающими напрямую использовать InterBase API
INTL	В каталоге находится динамическая библиотека gdsintl.dll, которая содержит международные наборы символов и информацию о сопоставлении международных символов (collation). Подробнее о поддержке международных (и русских в том числе) символов см. главу "Русификация InterBase" (ч. 1)
LIB	В этом каталоге находятся файлы *.lib, это библиотеки, которые будут полезны разработчикам баз данных, работающих с InterBase на Borland C++ и MSVC++
UDF	Важный каталог, который содержит динамические библиотеки, реализующие UDF (User Defined Functions). Библиотеки UDF для InterBase 6.x и всех его клонов должны находиться именно в этом каталоге. Подробнее об UDF см. главу "User Defined Functions" (ч. 1). При

Каталог или файл	Краткое описание
	установке в UDF записывается библиотека ib_udf.dll, которая содержит множество полезных функций
Файлы в корневом каталоге	
Ibconfig	Файл настроек InterBase-сервера. Позволяет влиять на производительность и работу сервера. Подробнее о возможных настройках сервера см. главу "Оптимизация работы InterBase" (ч. 4)
ibinstall.dll	Динамическая библиотека, реализующая IB Install API
InterBase.log	Файл протокола (лог), куда InterBase-сервер записывает все предупреждения и ошибки, возникающие в процессе работы. При возникновении любых проблем с работой СУБД InterBase следует обязательно просмотреть этот файл, а лучше всего это делать регулярно. Если проблем с базой данных нет, то лог увеличивается очень медленно и содержит в основном отметки о запуске guardian-процесса и завершении сервера. Если же размер InterBase.log растет очень быстро, то это свидетельствует о проблемах (возможно, скрытых) с базой данных или с аппаратным обеспечением
InterBase.msg	Файл содержит каталог сообщений о проблемах и ошибках, которые InterBase возвращает пользователю
Isc4.gdb	Это база данных пользователей данного InterBase-сервера
license.txt	Файл содержит лицензионное соглашение. Наличие этого файла необходимо для работы сервера
ReleaseNotes.pdf	Краткие замечания по той версии InterBase, которая у вас установлена. Файл в формате Adobe Acrobat Reader. Обычно содержит массу полезной информации об устанавливаемой версии, поэтому рекомендуется обязательно прочитать его

В таблице 4.16 приведен краткий обзор файлов и каталогов, входящих в установку InterBase SuperServer для Windows. Надо заметить, что данные в таблице приведены для случая полной установки сервера. Если же вы отказались во время установки от некоторых предлагаемых опций, то в своем установочном каталоге вы можете не наблюдать некоторые каталоги и файлы, описываемых в таблице. В связи с этим следует прояснить вопрос, что действительно критически важно для работы сервера, а чем можно пожертвовать. Ниже мы рассмотрим вопрос о минимально необходимом объеме установки сервера и клиента, а пока перейдем к подробному рассмотрению файлов, находящихся в каталоге BIN, содержащем основные исполняемые модули InterBase.

Каталог BIN в SuperServer

Мы рассмотрим только те файлы, которые относятся непосредственно к самому серверу. Если во время установки вы пожелали поставить ряд инструментов администратора и разработчика, например, таких, как IBConsole, то в каталоге Bin может оказаться больше файлов, которые связаны с этими дополнительными программами.

Таблица 4.17. Файлы в каталоге BIN IB SS под Windows

Файл	Краткое описание файла
ibguard.exe	Это guardian – процесс-хранитель InterBase. Его назначение – вновь запускать InterBase в случае сбоя. Надо отметить, что в Windows 2000 можно обойтись и без "гвардейца", так как ОС самостоятельно может перезапускать InterBase
ibserver.exe	Собственно это и есть сам InterBase-сервер. Firebird начиная с 1.5 называет свой основной файл fbserver.exe
ib_util.dll	Динамическая библиотека, которая содержит функции, расширяющие возможности InterBase, в частности функции управления памятью, которые необходимы для работы многих UDF
gbak.exe	Утилита командной строки, которая применяется для осуществления резервного копирования и восстановления из резервной копии. Подробнее о gbak см. главу "Резервное копирование и восстановление из резервной копии" (ч. 4)
gfix.exe	Утилита командной строки, применяющаяся для модификации и починки базы данных. Подробнее см. главу "Починка базы данных" (ч. 4)
gsec.exe	Утилита командной строки, применяющаяся для управления базой данных пользователей InterBase. Она дает возможность добавлять, удалять и изменять пользователей. Подробнее см. главу "Безопасность в InterBase: пользователи, роли и права" (ч. 4)
gstat.exe	Утилита командной строки, которая служит для сбора статистики по базе данных
iblockpr.exe	Утилита, которая анализирует таблицу блокировок. Информацию о таблице блокировок вы можете почерпнуть из приложения "Параметры конфигурационного файла InterBase"
isql.exe	ISQL – Interactive SQL. Интерпретатор SQL, реализованный как утилита командной строки, который может как работать в интерактивном режиме, так и выполнять файлы SQL-скриптов
gpre.exe	Препроцессор C, который будет полезен разработчи-

Файл	Краткое описание файла
	кам, использующим InterBase API
gdef.exe	Утилита, позволяющая создавать и изменять метаданные
gds32.dll	Динамическая библиотека, которая позволяет клиентам связываться с InterBase-сервером. Собственно эта библиотека и является клиентом InterBase
fbclient.dll	То же самое, что gds32dll в Firebird начиная с 1.5.
instreg.exe	Утилита командной строки, которая прописывает или удаляет из реестра Windows необходимые ключи, связанные с InterBase. Обычно эта утилита используется лишь программой-установщиком InterBase во время процесса установки и удаления InterBase
instsvc.exe	Утилита, которая используется для установки InterBase в качестве сервиса NT/2000. Обычно применяется только программой-установщиком InterBase.
qli.exe	QLI – Query Language Intepretator – интерпретатор языка GDML. Используется как для интерактивного выполнения команд GDML, так и для выполнения скриптов

Минимальный состав сервера InterBase SuperServer

Как видите, довольно большой список файлов. Просмотрев таблицу, можно кратко уяснить назначение каждого файла, однако вопрос "отделения зерен от плевел" остался открытым. Конечно, все перечисленные утилиты важны и являются неотъемлемой частью InterBase. Но какие же файлы содержат основную, базовую функциональность InterBase-сервера?

Вот список файлов, составляющих минимальный сервер InterBase архитектуры SuperServer под Windows, который сможет нормально функционировать (табл. 4.18):

Таблица 4.18. Минимальный набор файлов InterBase SS под Windows

Файл	Расположение	Описание
gds32.dll	InterBase\Bin	Клиентская библиотека InterBase
ibserver.exe	InterBase\Bin	Основной исполняемый файл InterBase (и всех клонов)
InterBase.msg	InterBase	Файл содержит сообщения сервера и клиента
msvcrt.dll	Windows\System или Winnt\System32	Динамическая библиотека Microsoft Visual C. Обычно уже имеется в ОС
isc4.gdb	InterBase	База данных пользователей InterBase
license.txt	InterBase	Файл лицензии. Для InterBase 6.x и его Open Source-клонов содержит

Файл	Расположение	Описание
		IPL
ib_license.dat	InterBase	Только для платных версий InterBase – сертифицированных билдов InterBase 6.x и версии InterBase 6.5

Надо отметить, что библиотека msvcrt.dll имеется на всех версиях Windows, кроме Windows 95, да и то в случае, если не установлено никаких пакетов обновления или хотя бы Microsoft Office. Файл сообщений InterBase.msg также не обязателен: если он будет отсутствовать, то некоторые сообщения об ошибках InterBase не будут корректно отображаться, но на работу сервера и клиентских приложений это не повлияет. Обратите внимание, что в случае необходимости поддерживать разнообразные кодировки необходимо включать в установку библиотеку gdsintl.dll! Без нее сервер InterBase работать будет, но без поддержки национальных языков.

InterBase Classic Server под Linux

Корневой каталог InterBase CS содержит несколько подкаталогов и файлов, которые описаны в таблице 4.19. Часть из них имеет то же самое название и назначение, что и в InterBase SS под Windows, поэтому подробно такие файлы описывать не будем.

Таблица 4.19. Состав InterBase CS для Linux

Каталог или файл	Краткое описание
/bin	Исполняемые модули InterBase, а также различные утилиты. См. ниже раздел "Каталог BIN для Classic Server"
/doc	Документация по InterBase – обычно содержит последние замечания, список исправленных и неисправленных ошибок и т. д.
/examples	Примеры использования InterBase API на C
/help	В этом каталоге находится база данных help.gdb, которая содержит краткую справку о командах и ключевых словах InterBase SQL
/include	Содержит заголовочные файлы для C, которые могут быть использованы, например, разработчиками на GNU C
/intl	Содержит gdsintl – библиотеку, содержащую информацию о кодировках (аналогично GDSINTL.DLL под Windows)
/lib	Каталог содержит клиентские библиотеки libgs.so и libib_util.so, которые являются аналогами gds32.dll и ib_ util.dll в Windows. Также в этом каталоге находится библиотека libgs.a, которая представляет собой библиотеку для статической сборки

Каталог или файл	Краткое описание
	клиента
/misc	Каталог содержит Firebird.xinetd – файл конфигурации для менеджера сервисов xinetd, в котором описаны параметры клона InterBase 6.x Firebird
/UDF	Каталог, в котором должны находиться UDF-библиотеки пользователя. По умолчанию содержит только библиотеку ib_udf
isc4.gdb	База данных пользователей InterBase
isc_config	Файл, хранящих настройки конфигурации для InterBase; аналогичен файлу ibconfig в версии InterBase под Windows
isc_event1.teststation	Файл, который содержит список событий. Используется менеджером блокировок
isc_lock1.teststation	Файл, который содержит таблицу блокировок. Используется менеджером блокировок
InterBase.log	Файл протокола InterBase
InterBase.msg	Файл сообщений InterBase
services.isc	Файл, который содержит информацию о соответствии номера порта имени сервиса, который будет использоваться для InterBase (обычно постановка в соответствие выглядит как gdsdb/tcp 3050). Эту постановку в соответствие необходимо добавить в файл /etc/services (обычно автоматически добавляется установщиком InterBase)

Рассмотрев вкратце состав InterBase Classic Server для Linux, рассмотрим теперь более подробно состав каталога BIN в этой версии. Он отличается в основном программными модулями, специфичными для архитектуры Classic.

Каталог BIN в InterBase Classic Server для Linux

Как будет ясно из главы "Classic и SuperServer", в Classic-архитектуре состав основных исполняемых файлов InterBase меняется – к нему добавляется менеджер блокировок и различные утилиты для управления InterBase. Файлы в каталоге Bin описаны в таблице 4.20:

Таблица 4.20. Файлы в каталоге Bin InterBase CS для Linux

Файл	Описание файла
changeDBAPassword.sh	Полезные скрипты на языке shell для некоторых действий: смены пользователя SYSDBA, смены пользователя, с правами которого запускается InterBase
CSchangeRunUser.sh	
CSrestoreRootRunUser.sh	
gbak	Утилита резервного копирования и восстановления

Файл	Описание файла
gdef	Утилита, позволяющая создавать и изменять метаданные
gds_drop	Утилита, которая останавливает InterBase
gds_inet_server	Основной исполняемый файл InterBase в Classic-версии InterBase
gds_lock_mgr	Менеджер блокировок
gds_lock_print	Утилита, применяющаяся для анализа таблицы блокировок
gds_pipe	Утилита, предназначенная для поддержки приложений, использующих POSIX-сигналы
gfix	Утилита модификации и восстановления базы данных
gpre	Препроцессор C для разработчиков на InterBase API
gsec	Утилита управления базой данных пользователей isc4.gdb
gsplit	Утилита для разделения/слияния одного большого файла базы данных в/из нескольких
gstat	Утилита для анализа статистики по базам данных InterBase
isc4.gbak	База данных пользователей InterBase
isql	Interactive SQL – утилита для ввода команд SQL и исполнения SQL-скриптов
qli	Query Language Interpretator – интерпретатор языка GDML

Заключение

В данной главе были рассмотрены две версии InterBase: Super под Windows и Classic под Linux, чтобы читатель получил представление о том, какой совокупностью файлов представлен InterBase. Состав файлов для InterBase обеих версий был рассмотрен на примере клона InterBase 6 – Firebird 1.0.

Classic и SuperServer

На данный момент существуют два варианта архитектуры InterBase, которые значительно отличаются друг от друга методами работы с клиентами, организацией взаимодействия собственных модулей и даже составом модулей, входящих в определенную реализацию архитектуры. Условно эти две различных архитектуры назвали Classic и SuperServer. Чтобы быстро войти в курс дела, коротко рассмотрим главные особенности этих архитектур.

Архитектура Classic кратко характеризуется следующей фразой: "каждому клиенту – собственный сервер". Это означает, что на каждое клиентское соединение на компьютере-сервере запускается серверный процесс, который обслуживает одного клиента. Сколько у нас будет клиентов, установивших соединения, столько экземпляров сервера запустится для их обслуживания (имейте в виду, что одна клиентская программа может открывать сколько угодно соединений с сервером).

Архитектуру SuperServer можно по аналогии охарактеризовать как "на всех клиентов – один сервер". Это означает, что все клиентские соединения обслуживаются одним серверным процессом, где каждым конкретным клиентом занимаются отдельные потоки (threads).

Сразу следует сказать, что компания Borland уже давно, еще до опубликования исходных кодов InterBase 6, заявляла о своей решимости полностью отказаться от архитектуры Classic и перейти исключительно на SuperServer, ввиду ее многочисленных достоинств.

Тем не менее Classic жива и поныне, имеет своих многочисленных приверженцев и не собирается так просто "сдаваться". Причины этой нешуточной схватки двух подходов мы сейчас рассмотрим, и начнем, конечно же, с исторического экскурса.

Ограничим глубину погружения в историю версией InterBase 4.x. Изначально InterBase 4 имел архитектуру Classic – это были версии 4.0 и 4.1. Версия 4.2 стала первым SuperServer в ряду продуктов InterBase. Версия InterBase 5.x уже не имела реализаций архитектуры Classic под платформу Windows – только SuperServer, но для Linux существует версия InterBase 5.6 с архитектурой Classic. В InterBase 6 сохраняется та же ситуация – под ОС семейства Unix/ Linux существуют InterBase 6 как в варианте SuperServer, так и в варианте Classic, а под Windows – только SuperServer.

Следует заметить, что деление на Classic и SuperServer не означает, что имеются два варианта исходных кодов для каждого вида архитектуры – один для Classic и другой для SuperServer (иначе со временем получились бы два разных сервера). Оба эти варианта архитектуры (и все реализации под разные ОС) производятся из общего набора исходных кодов с помощью директив Ifdef, разделяющих платформенно- и архитектурно-зависимые участки кода друг от друга. С помощью набора этих директив определяют, какой вариант и для какой платформы собирать. Естественно, для разных ОС сборка осуществляется с использованием разных библиотек ввода-вывода, управления памятью и т. д. Таким образом, начиная с версии InterBase 5.x компания Borland перестала разрабатывать вариант сервера под Windows с архитектурой Classic, в результате чего этот

вариант архитектуры доступен только поклонникам Unix/Linux-систем, а версии Classic под Windows ни в вариантах реализации от Borland, ни от Firebird не существует.

В конце 2001 года появился еще один альтернативный клон InterBase 6 – СУБД Yaffil, авторами которой являются петербургские программисты Олег Иванов и Алексей Карякин. Этот вариант InterBase как раз и имеет реализацию архитектуры Classic под Windows. Подробнее о Yaffil можно узнать в приложении "Yaffil – российский клон InterBase 6.x".

В версии Firebird 1.5 также появилась Classic-реализация, также претендующая на использование на мощных многопроцессорных серверах.

Стоит ли использовать классическую архитектуру или предпочесть SuperServer, – мы сейчас и попытаемся разобраться.

Classic

Рассмотрим подробнее архитектуру Classic-варианта сервера InterBase. В этой модели, как было сказано ранее, для каждого клиентского соединения запускается собственный серверный процесс, который обслуживает данного клиента. Процессом запуска управляет внешний процесс (это inetd или xinetd для Unix-систем).

Серверные процессы изолированы друг от друга. Как и любые другие процессы в ОС, они не могут свободно читать и писать друг у друга в памяти. Тем не менее работать они будут с одной базой данных, в результате чего могут возникнуть конфликты и рассогласования данных в базе данных. Представьте себе, что один серверный процесс пытается изменить страницу в базе данных, которую в данный момент изменяет другой процесс. Очевидно, что возникнет конфликт на почве распределения ресурсов. Чтобы сообщить о том, что определенные ресурсы в базе данных в данный момент используются и разрешить возникающие при "дележке" конфликты, существует специальный процесс – менеджер блокировок (gds_lock_mgr). Необходимость в менеджере блокировок возникает, когда второй клиент подсоединяется к базе. Именно в этот момент менеджер блокировок загружается в память, чтобы "следить за порядком".

Помимо разрешения конфликтов, существует дополнительная необходимость управления сервером в смысле администрирования. К сожалению, в Classic невозможно с клиента получить информацию о количестве клиентских соединений, обслуживаемых в данный момент сервером, так как для каждого клиента существует только один сервер, а информация об остальных серверных процессах, обслуживающих других клиентов, ему недоступна. Также в Classic-вариантах InterBase 6 и его клонов пока не реализовано Services API, которое позволяет управлять сервером через клиентские соединения, а не через специальные программы. Правда, надо отметить, что Yaffil Classic Server имеет реализацию Services API.

У каждого серверного процесса имеется собственный кеш, в котором хранятся используемые страницы базы данных. Например, если мы выделим на обслуживание каждого клиентского соединения 15 Мбайт кеша, то при 20 клиентах нам будет нужно 300 Мбайт ОЗУ только на кеш-память. Если предположить, что клиенты выполняют в основном какие-то однообразные запросы (а так оно

и есть в большинстве клиент-серверных систем), то будет очевидным многократное дублирование кешированной информации в каждом серверном процессе. Classic довольно расточителен: даже если клиенты выполняют абсолютно одинаковые запросы, все равно для каждого серверного процесса, обслуживающего одного клиента, будет кешироваться одна и та же информация.

Кроме кеша страниц базы данных, память отводится для кеширования схемы базы (метаданных). Каждый серверный процесс в архитектуре Classic будет иметь свою копию метаданных. На сложной базе (скажем, с сотнями таблиц и процедур) это может вылиться в десятки мегабайтов, причем отрегулировать этот размер нельзя.

Помимо вышеперечисленного, также велик расход ресурсов на запуск множества серверных процессов и функционирование менеджера блокировок. Чтобы преодолеть недостатки подхода "каждому клиенту – по серверу", была разработана архитектура SuperServer, на которую сейчас в компании Borland и направлены все усилия.

SuperServer

Архитектура SuperServer реализует принцип "все в одном", т. е. существует один-единственный серверный процесс, который обслуживает всех клиентов. Этот процесс никто не вызывает, он выполняется где-то в недрах ОС, ожидая запросов (на Unix-системах SuperServer реализован в виде демона, а в Windows NT /2000 – в виде службы Windows NT).

Все действия, выполняемые отдельными процессами в Classic-архитектуре, в в SuperServer выполняются отдельными потоками (threads) в рамках единого серверного процесса. Существует поток, который занимается разрешением конфликтов, другой поток обслуживает запросы на соединение, множество других потоков играют роль отдельных серверных процессов рахитектуры Classic по обслуживанию клиентов.

Из-за того что все клиенты находятся в адресном пространстве одного процесса (но в разных потоках), значительно упрощается и ускоряется разрешение конфликтов. Вместо механизмов межпроцессного взаимодействия используется более быстрый межпотоковый обмен. Исчезают издержки на запуск множества процессов, а ведь каждый процесс забирает у ОС довольно значительную часть ресурсов.

Для обработки пользовательских запросов в SuperServer применяется единый кеш, позволяющий свести к минимуму бесполезный расход памяти. Например, если клиент открыл соединение и дальше не проявляет активности, то ему будет выделен минимально необходимый для поддержания соединения размер ОЗУ.

Помимо вышеперечисленных достоинств, в SuperServer реализован интерфейс Services API, который позволяет управлять сервером через клиентское соединение. Таким образом, архитектура SuperServer способна обслужить больше клиентов, чем Classic, используя при этом меньше ресурсов.

Classic vs SuperServer

Как вы уже могли заметить, картина складывается довольно интересная: на каждый недостаток Classic у SuperServer находится достоинство. Classic расто-

чителен – SuperServer экономен, Classic без Services API – у SuperServer он есть. Однако, как и везде, здесь мы имеем "палку о двух концах", т. е., определенные недостатки Classic переходят в определенных ситуациях в его достоинства, а преимущества SuperServer превращаются в недостатки. Например, рассмотрим случай, когда у нас имеется, скажем, мощный двухпроцессорный компьютер-сервер с большим количеством ОЗУ, например 2 Гбайт.

Если мы установим на такую систему InterBase в варианте SuperServer, то будем наблюдать не ускорение, а замедление по сравнению с однопроцессорным вариантом того же сервера! Более того, с памятью будут твориться сплошные "недоразумения": экономный SuperServer будет "отказываться" от огромного ОЗУ, пытаясь всячески сэкономить оперативную память. Как же так, мощные процессоры, много памяти, а InterBase SuperServer не очень-то быстро работает?

Вот здесь и проявляются недостатки SuperServer. Проблему с масштабируемостью InterBase архитектуры SuperServer на многопроцессорных компьютерах давно признали в компании Borland. Дело в том, что ядро SuperServer не расчитано на использование нескольких процессоров.

При запуске множества потоков, обрабатывающих запросы клиентов, внутри серверного процесса SuperServer происходит следующее: ОС не может равномерно распределить время между потоками, потому что в InterBase активным может быть только один поток! Остальные *добровольно* ждут пока этот активный поток *сам* "отдаст" им процессор. Что остается ОС? Только выполнять этот единственный поток. В InterBase SuperServer встроен некоторый аналог планировщика потоков, реализующий *невытесняющую* многопоточность с *одним* активным потоком.

Итак, сервер InterBase SuperServer не может управлять распределением потоков по процессорам. В результате ОС при нарастании нагрузки начинает перебрасывать главный серверный процесс (ibserver.exe) с одного процессора на другой. На это тратятся системные ресурсы и время, что замедляет работу InterBase. С такой ситуацией на многопроцессорных системах борются путем "привязки" (affinity) InterBase варианта SuperServer к одному определенному процессору и игнорирования остальных.

Естественно, что приведенное выше описание является лишь аналогией для иллюстрации проблемы и не может служить точным описанием работы ядра SuperServer. Для точного описания механизмов работы следует обратиться непосредственно к анализу исходных кодов InterBase, что выходит за рамки этой книги.

Надо также отметить, что с распределением памяти у SuperServer тоже имеются некоторые проблемы. Если мы рассмотрим, как SuperServer обслуживает множество небольших клиентских запросов, то увидим довольно привлекательную картину: высокую производительность при относительно небольшом использовании оперативной и виртуальной памяти. Многочисленные клиентские запросы совместно (без дублирования) используют кешированную информацию SuperServer. Эта особенность делает вариант InterBase с архитектурой SuperServer особенно привлекательным для Web-приложений, ориентированных именно на такой стиль работы с базами данных.

Так как запросы небольшие, то они быстро отрабатывают и освобождают память для следующих за ними запросов.

Иная ситуация складывается, если постановка задачи требует наряду с простыми действиями по регистрации данных и просмотру данных, относящихся к какому-то документу или обозримому множеству документов, выполнения запросов аналитического характера, связанных со сканированием больших и сложных выборок и построением на их основе различных агрегатов.

Эти "тяжелые" запросы "проходятся" по большому количеству записей и требуют значительных ресурсов памяти и процессора для их выполнения. Мы пытаемся предусмотреть подобную ситуацию и используем мощное аппаратное обеспечение: высокопроизводительный компьютер-сервер с большим количеством ОЗУ. Однако, SuperServer "не понимает" нашей предусмотрительности и при выполнении "тяжелого" запроса пытается обращаться с ним как с небольшим, т. е. отдает ему доступную кеш-память и ресурсы, вытесняя при этом остальные запросы. Результат печален – пока выполняется запрос-тяжеловес, остальные запросы "топчутся в очереди". В связи с фактически последовательным обслуживанием потоков критическими участками кода ядра InterBase сервер просто не имеет другого выбора.

Остается сказать о достоинствах Classic, проявляющихся в этой ситуации.

Во-первых, масштабируемость архитектуры Classic на несколько процессоров. Из-за того что каждый клиент обслуживается независимым процессом, ОС спокойно "рассаживает" эти процессы по различным процессорам, динамически распределяя нагрузку при помощи системных средств управления приоритетами процессов, стоящих в очередь за использованием ресурсов процессора.

В результате действительно можно получить значительный выигрыш от многопроцессорной системы, соответствующий затратам на это оборудование.

Во-вторых, использование памяти и процессора при выполнении "тяжелых" запросов. Если мы запускаем какой-то очень интенсивно работающий с базой запрос, то он выполняется в рамках одного серверного процесса, обслуживающего данного клиента, не останавливая при этом остальные. Приоритет "тяжелого" запроса (фактически процесса) падает по мере увеличения времени использования им ресурсов процессора и он начинает "уступать дорогу" более приоритетным процессам других соединений, выполняющим короткие запросы, т. е. процессор занят на 90%, но на долю "долгожителя" приходится 80–70–60–50–40%… Он замедляет остальные, это заметно, но терпимо, и главное – у пользователя не возникает ощущения "подвешенности".

Вот где недостаток "избыточность" перетекает в преимущество "нагрузочная способность"!

Как бы то ни было, архитектура Classic значительно лучше SuperServer справляется с тяжелыми запросами при одновременном обслуживании нескольких клиентов и более корректно реализует вытесняющую многозадачность, что позволяет эффективнее справляться с запросами-"тяжеловесами".

Рекомендации по выбору архитектуры: Classic или SuperServer?

Прочитав предыдущий раздел, читатель может ощутить необходимость немедленно перейти на сервер InterBase с архитектурой Classic. Однако стоит побороть это иррациональное стремление и хорошенько все взвесить. Ведь нельзя сказать, что Classic (или SuperServer) – однозначно лучший выбор. У каждой архитектуры есть своя "весовая категория", в которой ее использование даст наилучшие результаты. Поэтому задача разработчика – правильно определить эту категорию.

Никакой разницы в структуре базы данных и в логике проектирования клиентских приложений при использовании различных вариантов архитектуры InterBase нет. При разработке можно спокойно воспользоваться SuperServer-сервером как наиболее экономичной версией InterBase. Это особенно важно, когда сервер стоит прямо на рабочей станции у разработчика.

Необходимость выбора той или иной архитектуры InterBase становится актуальной, когда разработка приложений базы данных близится к концу и готовая база данных и программное обеспечение, работающее с ней, готовятся к передаче заказчику.

Избежать больших проблем с выбором архитектуры позволяет нам гибкость InterBase. Выбирая любую из архитектур, можно быть уверенным, что этот выбор не является фатальным и его всегда можно изменить.

Итак, что выбрать? Для начала давайте условимся, что речь идет о действительно серьезных задачах, содержащих большое количество данных, для которых критична производительность. Ведь если речь идет о программе учета накладных, которой одновременно пользуется не более пяти человек, то выбор очевиден – это SuperServer. Это сэкономит ресурсы компьютера-сервера и даст отличную производительность. Можно сказать, что SuperServer – это выбор по умолчанию, т. е., если вы не знаете, что выбрать, выбирайте SuperServer, и скорее всего он удовлетворит потребностям 80% задач.

Но есть и такие 20 % задач, для которых критически важна масштабируемость при большом количестве обслуживаемых клиентов. Именно для них нужно осознанно производить выбор между архитектурами Classic и SuperServer.

Первым критерием выбора является выполняемая задача. Ответьте для себя на следующий вопрос: каково максимальное число клиентов будет единовременно подсоединено к вашей базе данных? Помните, что это число не равно числу пользователей системы, поскольку обычно даже в пиковые моменты одновременно подсоединяются к базе данных около 70–80 % от общего числа пользователей. Затем выясните, запросы какого характера выполняются в вашей системе. Это сильно зависит от того, ближе ли разработанный вами продукт к системе OLTP (on-line transaction processing), которая рассчитана на обработку множества небольших одновременных операций по занесению в базу данных, или же он ближе к системе DSS (decision support system), где преобладают длительные запросы, затрагивающие большое количество данных. В первом случае важно обработать множество запросов за короткий промежуток времени, во втором – гибко распределить нагрузку, чтобы сервер хорошо отрабатывал

"тяжелые" запросы, т. е. требуется не быстрота, а нагрузочная способность (быстро не надо: никто не требует быстроты от аналитических выборок и годовых отчетов).

Если у вас OLTP-подобная система, то добавляем плюс в пользу SuperServer, если DSS, то Classic. Также поступаем и с количеством активных пользователей: если это число более 50, то Classic становится однозначно оптимальнее – ведь скорее всего систему такого масштаба будет обслуживать высокопроизводительный многопроцессорный сервер.

Следующим критерием является оборудование. Если вы счастливый обладатель многопроцессорного сервера, то на этом муки выбора заканчиваются: однозначно следует выбрать Classic, вне зависимости от количества других плюсов и минусов. Classic в полной мере позволит ощутить преимущества нескольких процессоров.

Если процессор один, то надо продолжать выбирать. Сколько у вас оперативной памяти? Если больше 1 Гбайт, то ставим плюс Classic-архитектуре, если меньше – плюс SuperServer. Каков объем базы данных? Если он невелик, т. е. может 2–3 раза целиком поместиться в оперативной памяти компьютера-сервера, то следует поставить плюс SuperServer: его совместно используемый кеш позволит загрузить базу данных практически целиком в ОЗУ.

Теперь давайте подсчитаем плюсы. Большее количество у той или иной архитектуры следует рассматривать как возможный признак того, что она больше подойдет для вашей задачи. Но в любом случае (за исключением многопроцессорных систем, где однозначно выигрывает Classic) необходимо протестировать производительность обеих архитектур с вашей базой данных на конкретной конфигурации аппаратного обеспечения (тому, как настроить и оптимизировать ваш конкретный компьютер-сервер и функционирующий на нем InterBase, посвящена глава "Оптимизация работы InterBase" (ч. 4)). Никто не может заранее предсказать, какая именно архитектура станет наилучшим решением для вашей конкретной системы СУБД, но тот факт, что у вас есть выбор, представляет собой несомненное достоинство InterBase… для тех, кто сумеет воспользоваться этим выбором.

Структура базы данных InterBase

Физическая структура базы данных

Зачем изучать физическую структуру базы данных?

Говоря о физической структуре базы данных InterBase, обычно подразумевают то, что представляют собой данные с точки зрения низкоуровневой организации данных – вплоть до уровня байтов. Многие программисты, работающие на языках высокого уровня, пренебрегают изучением физической структуры. Однако знание основных принципов организации информации внутри базы данных дает ключ к действительно эффективному проектированию приложений баз данных. Поэтому в этой главе мы проведем экскурс во "внутренности" устройства базы данных InterBase и разберемся в том, как она устроена.

Итак, для чего предназначена система управления базами данных? Очевидно, для хранения и управления данными. Это звучит банально, но об этом стоит задуматься. Пользователь некоторым образом помещает данные в СУБД, которая эти данные каким-то образом переводит в понятные ей внутренние форматы. Вы можете представлять себе "нолики и единички", если слова "внутренний формат данных" вызывают какие-то трудности с ассоциациями. СУБД хранит эти данные и по первому требованию должна извлечь их из своего формата, преобразовать в удобочитаемый вид и предоставить пользователю.

Предметом рассмотрения этой главы будет то, как именно СУБД хранит свои данные, в каком виде, как они организованы на диске. Мы попробуем прояснить, как из битов и байтов, лежащих на диске, получается ценная информация, помещаемая в базу данных пользователями.

Файлы базы данных InterBase

Все данные, которые пользователи "помещают" в базу, используя любой инструмент из множества применяемых для этой цепи "складируются" сервером в некую сущность – базу данных. Обычно под *базой данных* понимается и сам сервер СУБД, и пользовательская информация, и даже клиентские программы, которые работают с данными. Мы будем понимать в этой главе под базой данных совершенно конкретную вещь – файлы базы данных.

База данных InterBase представляет собой один или несколько файлов, в которых находится информация обо всем, что связано с этой базой. Исключение составляет информация о пользователях, поскольку пользователи определяются на уровне всего сервера и хранятся отдельно, в системной базе данных ISC4.GDB. Внутри файлов базы данных содержится вся информация о базе: сами данные, индексы, триггеры, хранимые процедуры и т. д.

База данных InterBase для среднего проекта представляет собой один файл, так как ограничение в 32 Гбайта на размер одного файла базы данных позволяет держать все данные в одном файле (версии ниже InterBase 6.5, Firebird 1.0 и Yaffil 1.0 имеют ограничение в 2-4 Гбайт, в зависимости от ОС). 32 гигабайт вполне хватает для хранения информации приложения баз данных среднего размера. Но при необходимости можно разбить базу данных на несколько файлов. Известны базы данных InterBase размером в сотни гигабайт.

IBSurgeon – проводник по базе данных InterBase

Так как нам необходимо подробно разобраться в строении файлов баз данных InterBase, то желательно иметь какой-нибудь удобный инструмент, позволяющий работать с файлами базы данных напрямую, а не через ядро сервера InterBase. Самый простой способ – это воспользоваться обычным шестнадцатеричным просмотрщиком и попытаться разобраться в структуре файлов базы данных, рассматривая ее HEX-представление. Это было бы довольно утомительное занятие.

Но, к счастью, существует инструмент для прямой работы с базами данных InterBase, а также всех его клонов – Firebird и Yaffil. Это *IBSurgeon (www.ibsurgeon.com)* – инструмент для непосредственной низкоуровневой работы с базами данных InterBase/Firebird/Yaffil, который может использоваться для исследования внутренней структуры баз данных InterBase и диагностики поврежденных баз данных с целью их восстановления. (Подробности см. в приложении "Инструменты администратора и разработчика InterBase").

IBSurgeon использует собственный альтернативный механизм доступа к базам данных InterBase/Firebird/Yaffil, что позволяет диагностировать базы данных в любом состоянии, в том числе и те, которые не открываются с помощью ядра сервера InterBase/Firebird/Yaffil.

Мы воспользуемся IBSurgeon для того, чтобы проиллюстрировать внутреннее строение базы данных и придать ему видимые, "реальные" очертания.

Файлы *.GDB изнутри

GDB – это расширение, которое рекомендуют использовать для файлов баз данных InterBase. Первое, что нужно сказать о строении GDB-файла, – это то, что он представляет собой набор *страниц* жестко определенного размера. Размер файла базы данных кратен размеру страницы, который неизменен для всех файлов данной базы данных. Разные версии InterBase поддерживают различные размеры страниц, что отражено в таблице 4.21. Размер страницы задается при создании базы данных и не может быть изменен в течение ее жизненного цикла, т. е. изменить размер страницы возможно только при создании базы из резервной копии (restore).

Таблица 4.21. Размер страницы, поддерживаемый различными версиями InterBase

Версия InterBase	Размер страницы, байт				
	1024	2048	4096	8192	16384
InterBase 4.0	*	*	*	*	
InterBase 5.x	*	*	*	*	
InterBase 6.0x	*	*	*	*	
Firebird 1.x/Yaffil 1.x /InterBase 6.5 и выше	*	*	*	*	*

Чтение и запись данных в базе данных осуществляется постранично, и многие важные характеристики базы данных и сервера, такие, например, как размер буфера базы данных (Database cache), зависят от размера страницы и исчисляются в "страницах".

Давайте откроем какую-нибудь базу данных InterBase с помощью IBSurgeon. Для этого достаточно дважды щелкнуть по файлу базы данных. На рисунке 4.2 изображен список страниц, который показывается после того, как IBSurgeon открыл базу данных.

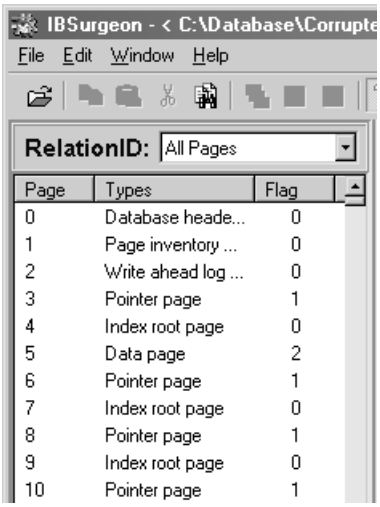


Рис. 4.2. Список страниц базы данных

Страницы бывают различных *типов*, каждый из которых служит определенной цели. Взаимозависимости различных типов страниц условно представлены на рис. 4.3. Он схематично изображает расположение страниц в файле базы данных – слева направо, сверху вниз, если считать от начала файла. Страницы одного типа не идут строго одна за другой – они могут быть перемешаны свободно, располагаясь в файле в том порядке, в котором они создавались сервером при расширении или создании базы данных.

Некоторые типы страниц выглядят "болтающимися без дела", т. е. не имеющими ссылок на другие типы страниц. Однако здесь нет никакого противоречия, просто эти типы страниц связаны и используются на другом структурном уровне, они могут связываться с помощью таблицы RDB$PAGES и других системных таблиц (эта таблица и другие системные объекты будут рассмотрены ниже, в разделе "Логическая структура базы данных"). На рис. 4.3 изображены только явные ссылки между страницами на физическом уровне.

Рассмотрим подробнее, какие бывают типы страниц в базе данных InterBase. В файле ods.h из набора исходных кодов InterBase находится информация обо всех возможных типах страниц. К этому файлу мы будем часто обращаться, чтобы из первоисточника получить данные не только об ODS, но и о многих других основополагающих вещах ядра InterBase.

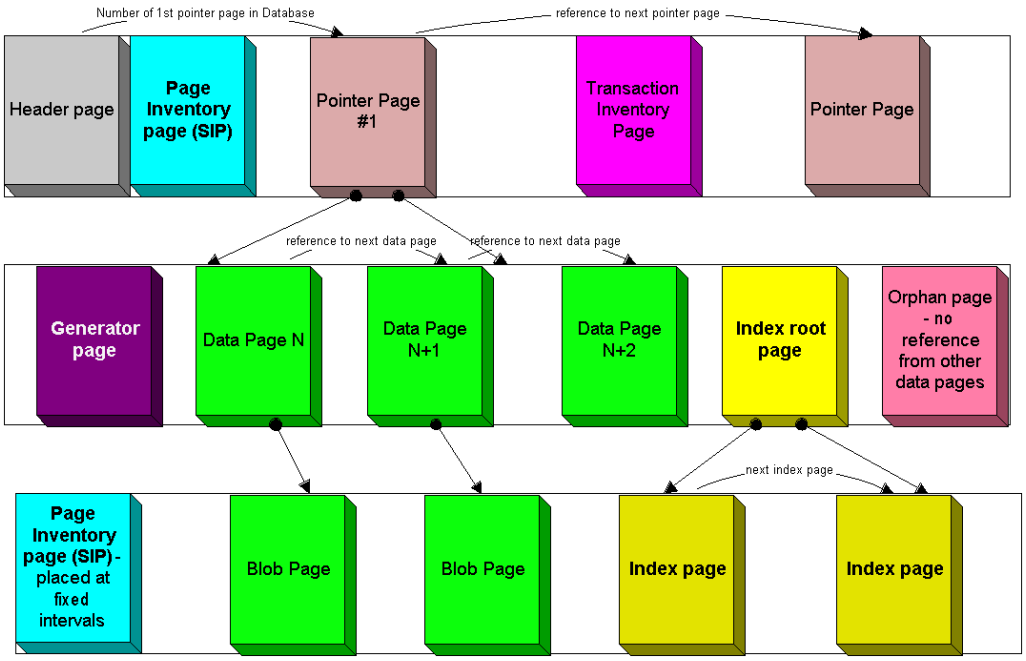


Рис. 4.3. Взаимозависимости между различными типами страниц в базе данных InterBase

Всего задекларировано 11 типов страниц, однако достойны объяснения лишь 9 из них, что ясно видно из табл. 4.22. Типы страниц с идентификаторами 0 и 10 не определены или не используются.

Таблица 4.22. Типы страниц

Определение в ods.h	Идентификатор страницы	Английское название страницы	Русская интерпретации английского названия
pag_undefined	0	Undefined - If a page has this page type it is probably free	Неопределенный тип страницы – возможно, страница свободна
pag_header	1	Database header page	Страница заголовка базы данных
pag_pages	2	Page inventory page (or Space inventory page - SIP)	Страница, хранящая информацию о распределении страниц
pag_transactions	3	Transaction inventory page (TIP)	Страница учета транзакций
pag_pointer	4	Pointer page	Страница указа-

Определение в ods.h	Идентификатор страницы	Английское название страницы	Русская интерпретации английского названия
			телей
pag_data	5	Data page	Страница данных
pag_root	6	Index root page	Страница вершины индекса
pag_index	7	Index (B-tree) page	Страница индексов
pag_blob	8	Blob data page	Страница для хранения BLOB-данных
pag_ids	9	Gen-ids	Страница генераторов
pag_log	10	Write ahead log information	Не используется

Каждая из страниц имеет заголовок, содержащий информацию о типе страницы и номер следующей страницы такого же типа. Полный список параметров, содержащихся в заголовке каждой страницы, можно получить, рассмотрев структуру pag в файле определений ods.h. Желающие досконально разобраться в работе со страницами могут обратиться к этому и другим определениям в этом файле.

Типы страниц и их использование

Давайте подробнее рассмотрим каждый тип страниц и ознакомимся с их назначением и информацией, которая на них содержится. Начнем по порядку – с самой первой страницы.

Любая работа с базой данных начинается с чтения страницы заголовка базы данных (или заголовочной страницы). **Страница заголовка** базы данных всегда идет первой в первом файле базы данных. Соответственно она изображена первой и на рис. 4.3 (если представить, что рисунок представляет протяженность файла базы данных слева направо, сверху вниз).

Заголовочная страница содержит информацию о базе данных в целом. На рис. 4.4 изображена страница данных так, как это показывает нам IBSurgeon.

Получить представление о содержании заголовочной страницы также можно, получив статистику по базе данных. Для этого можно воспользоваться утилитой командной строки *gstat* или каким-нибудь более удобным инструментом для администрирования InterBase из списка в приложении "Инструменты администратора и разработчика InterBase". Подробнее процесс получения статистики для базы данных и описание данных заголовочной страницы содержатся в главе "Статистика в InterBase" (ч. 4).

Здесь стоит лишь отметить, что заголовочная страница содержит такую важную информацию, как размер страницы, номер версии ODS (On-Disk Structure)

(см. о ней ниже), контрольная сумма базы данных (для On-Disk Structure 9.xx и старше она обычно равна 12345), дата создания базы данных, а также информацию о транзакциях и множество других сведений. Например, Implementation ID хранит информацию о том, под какой ОС эта база данных была создана.

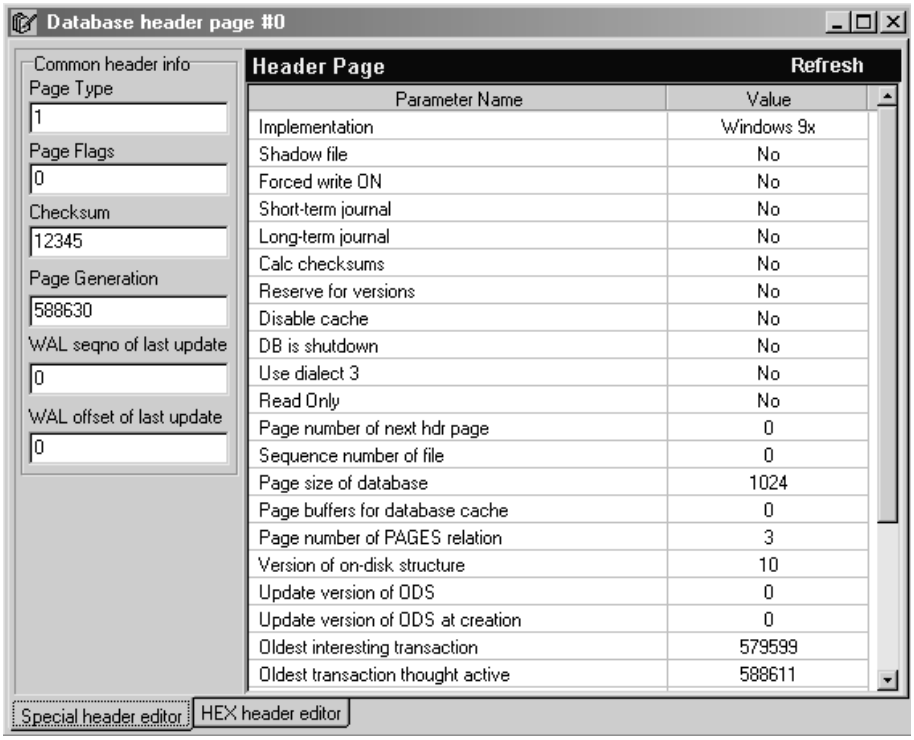


Рис. 4.4. Страница заголовка базы данных (header page)

При подключении к базе сервер InterBase считывает первые 1024 байта информации от начала файла и определяет по считанным значениям, действительно ли файл, указанный в строке соединения, является базой данных InterBase. Затем сервер с заголовочной страницы считывает номер версии ODS (On-Disk Structure) и размер страницы в этой базе данных и, если версия ODS совместима с реализацией сервера, производит перечитывание всей заголовочной страницы, пользуясь уже корректным размером страницы, полученным из первых 1024 байт. После этого с заголовочной страницы считываются остальные важные параметры базы данных – режим чтения-записи, диалект базы данных и т.д.

На заголовочной странице находится ссылка на первую из страниц указателей (Pointer page), на которой содержащие ссылки на страницы данных, содержат метаданные: таблицу RDB$Pages (см. ниже в разделе "Логическая структура базы данных InterBase"). На рис. 4.3 эта ссылка проиллюстрирована стрелкой с надписью Number of the 1st pointer page in database, т. е. "номер первой страницы указателей в базе данных". Сервер читает номер первой страницы указателей

с заголовочной страницы и переходит к ней. Фактически он просто отсчитывает "размер страницы умножить на номер страницы" байт.

Страница указателей состоит из упорядоченного массива номеров страниц данных, составляющих определенную таблицу (таблица понимается в смысле SQL-объекта, описываемого логической структурой базы данных). Вот как интерпретирует страницу указателей IBSurgeon (рис. 4.5).

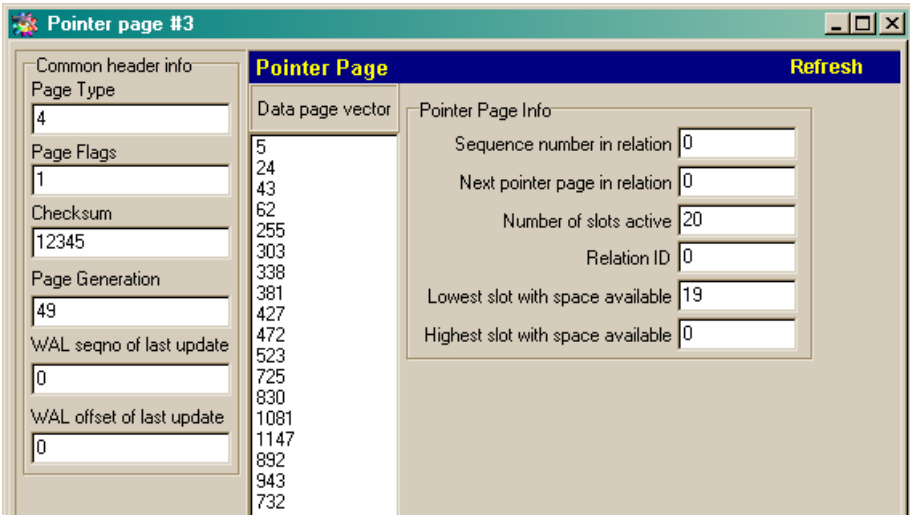


Рис. 4.5. Страница указателей базы данных InterBase (pointer page)

Страница содержит список страниц данных (data page vector), которые составляют определенную таблицу в базе данных. Список представляет собой массив указателей, соответствующих номерам страниц данных в файле.

Сервер считывает 4-байтовый номер страницы данных и переходит к нужной странице данных. Перейдя на первую страницу данных, составляющих RDB$Pages, сервер начинает построение внутреннего представления базы данных, которое в дальнейшем используется сервером для всех операций с базой данных.

В RDB$Pages содержатся ссылки не только на страницы данных, хранящие информацию о базе данных, но и на все остальные страницы, играющие роль в обеспечении работы базы данных.

> Мы часто упоминаем эту таблицу, что, строго говоря, относится к логической структуре базы данных. Тем не менее все очень взаимосвязано, поэтому нельзя описывать одно, не ссылаясь на другое.

Важным типом страниц являются **страницы, учитывающие транзакции** (TIP, Transaction inventory page). Они, как и все страницы, состоят из заголовка и основной части, представляющей собой массив из 2-битовых последовательностей. Эти последовательности описывают состояние транзакций в базе данных (подробности о транзакциях см. в главе "Транзакции. Параметры транзакций" (ч. 1)).

Таблица 4.23. Возможные состояния транзакции в Transaction inventory page

Значение последовательности на странице учета транзакций	Смысл
0	Транзакция не запускалась, активна или потеряна без commit или rollback
1	Транзакция завершилась Commit
2	Транзакция завершилась rollback
3	Limbo-транзакция (для 2PC)

Каждой версии записи соответствует свой идентификатор транзакции, что позволяет одновременно выполняющимся транзакциям "узнавать" о состоянии друг друга и разрешать конфликты при многопользовательской работе.

Заголовочная страница базы данных, страницы указателей и страницы учета транзакций относятся к служебным ("housekeeping") типам страниц, которые используются только сервером. Информация, содержащаяся в них, никогда не попадает к пользователям InterBase. К служебному типу страниц также относятся **страницы учета страниц** (обычно они упоминаются как Page Inventory Page (PIP) или Space Inventory pages (SIP)). Эти страницы расположены начиная со второй, т. е. первая PIP следует прямо за заголовочной страницей, и появляются в базе данных через фиксированные промежутки страниц других типов. Размер этих промежутков, т. е. через какое количество страниц других типов появляется PIP, зависит от размера страницы, установленного для данной базы данных. Page Inventory Pages не учитываются на страницах указателей (Pointer page) и не указаны в RDB$Pages. Целостность этих страниц критична для "здоровья" всей базы, так как содержимое PIP описывает *состояние* всех остальных страниц в базе данных. Каждая страница базы данных может иметь 3 состояния: *нераспределенное* (not allocated), *распределенное* (allocated with space), *распределенное и заполненное* (allocated and full). Когда появляется необходимость в дополнительном пространстве для новых данных, сервер проверяет PIP на предмет наличия нераспределенных страниц. Если находится такая страница, сервер изменяет ее состояние на "распределенное". Если нераспределенных страниц нет, то база данных расширяется – к ней дописывается новая страница данных.

Изображение страницы данных в IBSurgeon и содержащихся на ней данных приведено на рис. 4.6.

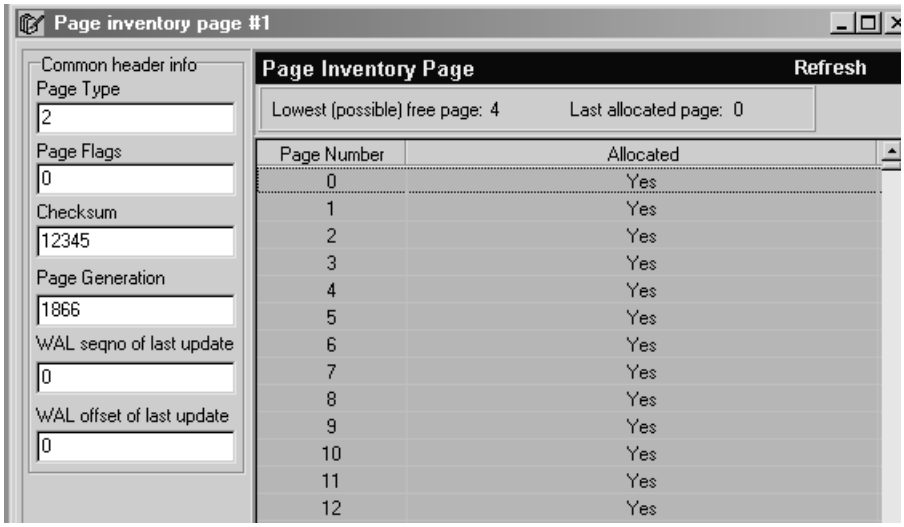


Рис. 4.6. Страница учета страниц (PIP)

Как только страница распределена, InterBase записывает ее состояние на SIP, а затем пишет саму страницу. После этого необходимо присоединить вновь образованную страницу к какому-нибудь множеству страниц, например к страницам данных для какой-то таблицы. Для этого надо записать ссылку на эту "свежую" страницу на последней странице этого множества страниц – например, на последнюю страницу данных какой-то таблицы.

Если сервер прервал свою работу сразу после записи на SIP, но не дойдя до записи ссылки на страницы, которая ссылается на только что распределенную, то эта только что распределенная страница становится "*потерянной*" *(orphan)*. Потерянная страница физически создана, зарезервирована на SIP, но ссылок с других страниц на нее нет, а значит, сервер не сможет найти ее и записать на нее данные. Потерянная страница изображена красным квадратиком на рис. 4.3. Потерянные страницы чаще всего возникают в результате неожиданного выключения питания у сервера и "лечатся" специальным инструментом для починки баз данных *gfix* (смотри раздел "Починка базы данных").

Прежде чем перейти к рассмотрению *страниц данных*, следует упомянуть о важных типах страниц: **страницах генераторов** и **страницах индексов**. Страницы генераторов представляют собой массив 4-байтовых чисел, которые показывают состояние генераторов. Фактически генератор – это обыкновенный счетчик.

На рис. 4.7 показана страница генераторов. Обратите внимание, что, хотя IBSurgeon и показывает имена генераторов, это не значит, что эти имена хранятся на страницах генераторов, – это сделано для удобства пользователя, исследующего базу данных. На самом деле имена генераторов хранятся в системной таблице RDB$Generators.

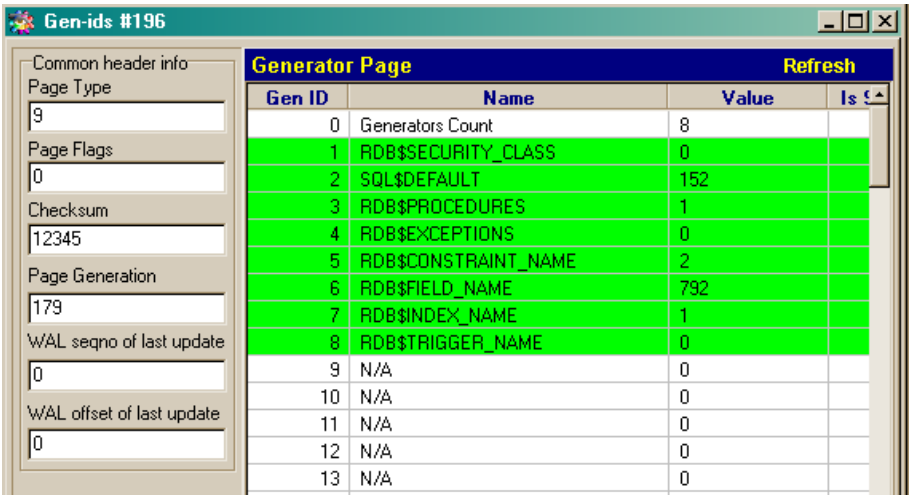


Рис. 4.7. Страница генераторов (gen-ids)

Как видите, в данном примере в базе данных содержатся только системные генераторы, начинающиеся с префикса RDB$. (О назначении и использовании генераторов при разработке приложений баз данных InterBase рассказано в главе "Таблицы. Первичные ключи и генераторы" (ч. 1)). Страницы генераторов учитываются наряду с другими страницами в таблице RDB$Pages.

Каждой таблице, вне зависимости от того, имеет ли она индексы или нет, соответствует по крайней мере одна **страница вершины индекса** (index root page). Эта страница содержит указатели на страницы индексов для соответствующей таблицы. Можно сказать, что index root page играет для страниц индексов такую же роль, какую играет страница указателей для страниц данных. Поэтому IBSurgeon показывает ее сходным образом.

Изображение страницы вершины индекса приведено на рис. 4.8.

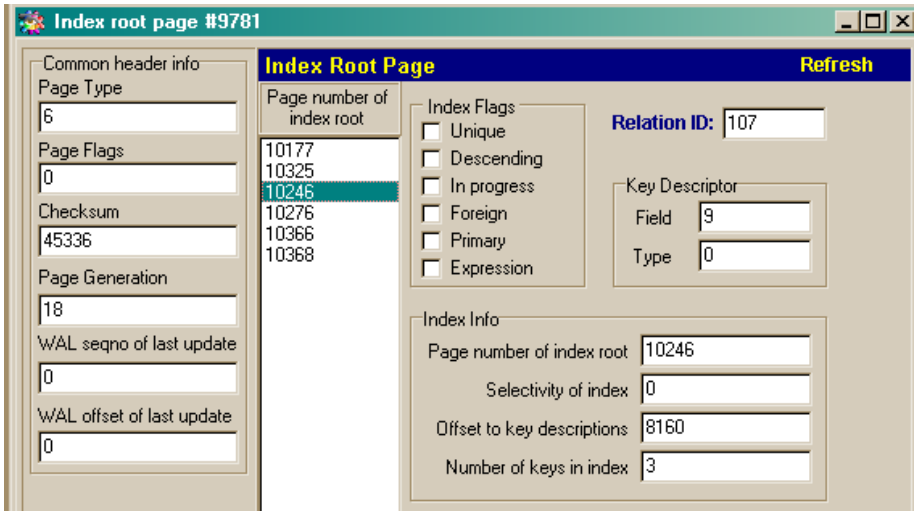


Рис. 4.8. Страница вершины индексов (index root page)

Страница вершины индексов содержит список страниц, на которых собственно хранятся сами значения индексов, а также системную информацию об индексах – о селективности индексов и различных флагах. Подробнее про индексы, о их роли и значениях в базах данных рассказано в главе "Индексы" (ч. 1.).

Непосредственно значения индексов содержат **индексные страницы** (index pages). Пример такой страницы изображен на рис. 4.9.

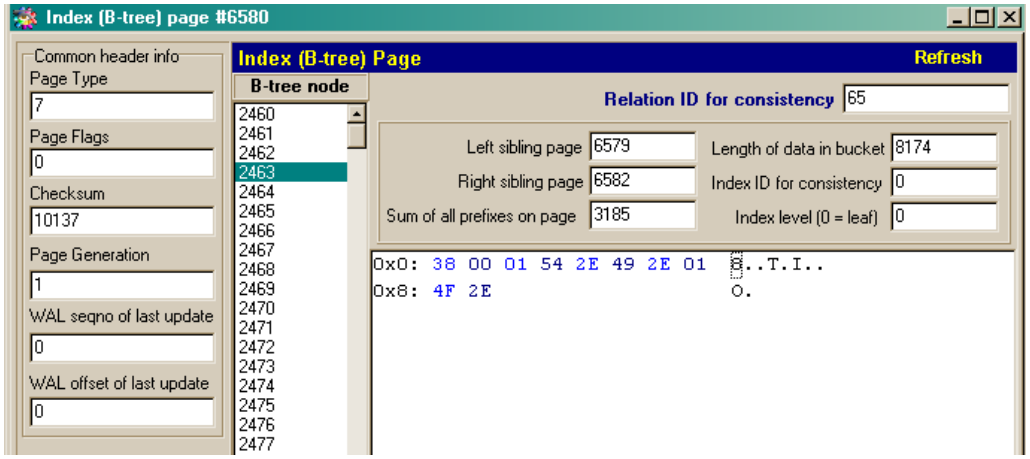


Рис. 4.9. Страница индексов (index B-tree page)

Страница индексов хранит упакованные значения проиндексированных данных. Используется достаточно сложный механизм индексации, особенно при построении составных индексов (включающих в себя несколько полей).

Пользовательскую информацию в основном хранят **страницы данных** (data pages) и страницы, содержащие BLOB-значения (Blob pages). Страницы данных содержат записи в пользовательских таблицах базы данных, фрагменты записей, старые версии записей, различия между версиями, BLOB-поля (если они помещаются на странице) и т. д. Что касается Blob-полей, то они связаны с записями на страницах данных и содержат данные большого размера, не помещающиеся на странице данных. Ссылочный тип хранения BLOB-значений позволяет хранить очень большие данные.

Изображение страницы данных в IBSurgeon приведено на рис. 4.10:

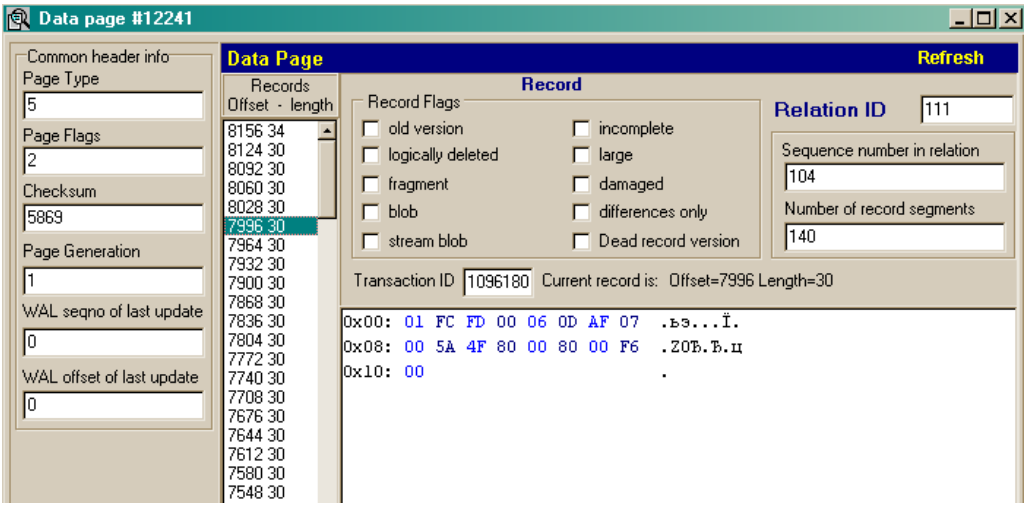


Рис. 4.10. Страница данных (data page)

Заголовок страницы данных содержит тип страницы (Page Type), идентификатор таблицы (RelationID), информацию о которой содержит страница, а также номер следующей страницы с данными в этой таблице.

Записи хранятся на страницах данных с конца страницы и по мере заполнения размещаются ближе к началу страницы.

В этом легко убедиться, взглянув на список записей на странице (row indexes), содержащий два значения – сдвиг записи (Offset) на странице и ее длину (length). Как видите, в начале списка располагаются записи, находящиеся на самом конце страницы – например, первая запись имеет сдвиг 8156 байт, а длину 34 байта, следовательно, заканчивается она 8156+34=8192 байтом – на самом краю страницы (в данном случае размер страницы – 8192 байта, т. е. размер страницы – 8192 байта).

Когда страница заполнена (сверху – данными, снизу – индексами записей), сервер начинает записывать новые записи и версии старых записей на новые страницы.

Исходя из описанного механизма заполнения страниц, можно легко сказать, почему специалисты по InterBase так настойчиво рекомендуют использовать страницы данных большого размера (4096 байт как минимум, а лучше 8192 бай-

та). Если мы создадим таблицу, одна запись которой будет иметь значительный размер (например, 10 полей VARCHAR(255)), то они будут занимать, будучи заполненными, более 2550 байт. Т. е. одна такая запись не поместится на странице малого размера (1024 или 2048 байт). Очевидно, что необходимость загрузить несколько страниц с диска, чтобы прочитать единственную запись, не ускорит работу с вашей базой данных. Таким образом, настоятельно рекомендуется переопределять размер страницы данных при создании или восстановлении базы данных, потому что по умолчанию устанавливается размер 1024 байта.

Кратко рассмотрев основные типы страниц данных в InterBase и их назначение, перейдем теперь на более высокий структурный уровень.

Понятие об ODS

ODS – это аббревиатура для On-Disk Structure, т. е. структура данных баз данных InterBase на диске. ODS определяет, как организованы данные внутри файлов базы данных. Определение основных констант и структур данных для реализации On-Disk Structure находится в файле из комплекта исходных кодов ods.h.

ODS в процессе развития InterBase менялась, и, чтобы сервер при работе с конкретной базой данных точно знал, с чем он имеет дело, он выясняет номер версии ODS. Файл Ods.h представляет нам следующую картину версий On-Disk Structure:

- ODS 5 поставлялась с версией InterBase 3.3 и более не поддерживается;
- ODS 6 и ODS 7 никогда не выходили в свет;
- ODS 8 поставляется с версией InterBase 4.0;
- ODS 9 поставляется с версией InterBase 5.x и выше;
- ODS 10 начала поставляться с InterBase 6.
- ODS 11 появилась в InterBase 7.0, 11.1 – в 7.1

Помимо основных (major) версий ODS существуют еще вспомогательные (minor) версии, которые зависят от конкретной версии сервера базы данных, создавшего их. Основные номера версии записываются в целой части числа, которое обозначает версию, а минорные – в дробной. Например, версия сервера 4.0 создает базы данных, которые имеют ODS 8.0, а InterBase 4.2 – 8.2. Переход между минорными версиями "снизу вверх" осуществляется автоматически. Например, достаточно открыть базу с ODS 8.0, созданную сервером 4.0, при помощи InterBase 4.2 – и ODS этой базы будет иметь версию 8.2. Переход между основными версиями базы данных осуществляется только через резервное копирование (backup) базы данных с применением старой версии сервера и восстановление из резервной копии (restore) с использованием новой версии сервера. Процесс перехода между версиями подробно рассмотрен в главе "Миграция" (ч. 4).

Важным моментом в реализации поддержки ODS для версий InterBase 4.x и 5.x является совместимость серверов InterBase 4.x и InterBase 5.x с версией, на единицу меньшей, чем реализация конкретного сервера. InterBase поддерживает несколько возможных ODS и, в зависимости от ее версии ODS, при присоединении к конкретной базе данных выбирает поддержку нужной реализации

ODS. Механизм принятия решения о том, какую реализацию поддержки ODS выбрать в данном конкретном случае, называется Y-Valve (автор Стива Тентона). Проще говоря, базу с ODS 8.x ("родной" для InterBase 4.0), можно открыть в InterBase 5.x и работать с ней.

Совместимость версий ODS идет "сверху вниз", т. е. сервер с более высокой версией и все его инструменты сумеют работать с базой данных, созданной ранними версиями сервера, но никак не наоборот. Попытавшись открыть базу, созданную в 6-й версии InterBase, при помощи InterBase 5.x, вы получите ошибку "Unsupported On-disk structure: Found ODS 10, supported ODS 9".

Также следует заметить, что бета-версия InterBase 6.0 поддерживала только ODS 10. В то же время Firebird 1.0 уже позволяет открывать базы как с ODS 8.xx, так и с 9.xx. Следует обратить внимание, что Firebird позволял только открывать базы данных с ODS 8 и 9, потому что гарантировать что-то при таком способе работы ничего нельзя. Т. е., несмотря на совместимость между версиями ODS "сверху вниз", перенос базы данных между версиями сервера лучше производить документированным способом.

> Описание перехода между версиями снизу вверх и обратно смотрите в главе "Миграция" (ч. 4).

Особенно важна версия ODS в вопросах, связанных с резервным копированием и извлечением базы данных, а также с восстановлением испорченных баз данных. Инструменты резервного копирования *gbak* и восстановления *gfix* следят за версией ODS и просто не станут работать, если версия ODS-базы, которую они должны обслуживать, больше версии, реализованной в них. Это значит, что gbak от 4.x не сможет создать резервную копию базы данных, если она создана сервером 5.x, однако наоборот – пожалуйста.

Мост между физической и логической структурой базы данных

Мы рассмотрели в общих чертах физическую структуру файлов базы данных. Теперь надо перейти к логической структуре базы данных. Чтобы переход произошел без каких-то "предельных переходов" в понятиях, оставив после себя ощущение непонятности материала и желание перечитать главу с самого начала, давайте построим некий логический "мост" между физическим и логическим уровнями представления информации в базе данных.

Все, что хранится на различных страницах базы данных, необходимо как-то организовать в памяти компьютера, преобразовать данные из файла базы данных в совокупность внутрисерверных объектов и переменных. Эта совокупность называется *внутренним образом базы данных (internal database image)*, по терминологии Анн Харрисон [10].

Итак, попробуем рассмотреть процесс построения внутреннего образа базы данных.

- Сервер читает 1024 байта из начала файла и, если это действительно файл базы данных InterBase, определяет размер страницы этой базы и перечитывает всю заголовочную страницу целиком.

- С заголовочной страницы сервер извлекает номер страницы указателей, на которой хранятся ссылки на страницы данных, определяющие таблицу RDB$Pages.
- Сервер переходит к этой странице указателей и начинает считывать из указанных там страниц данных информацию. Он заполняет данными первую таблицу RDB$Pages. Эта таблица является чем-то вроде мостика между физическими объектами – страницами файлов базы данных и логическими – таблицами. Структура RDB$Pages, как и других системных таблиц, жестко "прошита" в InterBase.
- Получив данные о распределении страниц по *отношениям* (relations, в сущности, это то же самое, что и обычные таблицы, и для упрощения можно мысленно подменять эти понятия), InterBase начинает формировать структуры данных: сначала системные таблицы, ограничения и индексы, а потом уже пользовательские объекты.
- После инициализации системных и пользовательских *метаданных* (так называют таблицы, ограничения, индексы и все остальные объекты базы данных), InterBase возвращает пользователю, попросившему открыть базу данных, handle этой базы данных (некоторые используют термин "рукоятка" или просто транскрипцию английского слова – "хэндл"). В сущности, handle – это некоторый идентификатор, который указывает InterBase, с какой именно базой данных работать, поскольку одновременно могут работать несколько пользователей, а значит, могут быть открыты несколько баз данных.
- После этих операций база данных считается открытой и сервер готов выполнять запросы пользователей к ней.

Теперь, когда проложен некоторый "мост", связывающий физическую и логическую структуру базы данных, можно переходить к особенностям логической структуры.

Логическая структура базы данных InterBase

Логическая структура – понятие достаточно расплывчатое, поэтому мы попробуем постепенно освоить ключевые идеи, надеясь, что позже они создадут интуитивно понятное ощущение. Первое, что мы рассмотрим из относящегося к логической структуре базы данных, это *системные таблицы* и их содержимое.

Системные таблицы описывают *метаданные***,** как системные, так и пользовательские. Вообще говоря, термин "метаданные" означает "данные, описывающие множество данных". Приставка "мета-" означает "описывает множество". Например, метаязык – это язык, описывающий множество языков. Метаданные описывают пользовательские данные, т. е. таблицы, триггеры, представления, хранимые процедуры и т. д. – все, что реализует правила хранения и обработки той информации, ради хранения и обработки которой и создается конкретная база данных.

Довольно забавно в первый раз узнать, что все метаданные, – как пользовательские таблицы, триггеры, представления, так и все системные объекты, – хранятся в таких же точно таблицах, из которых можно читать и писать данные

с помощью обычных SQL-запросов. Эти таблицы "визуально" отличаются только тем, что их имена начинаются с RDB$. Эти 4 символа зарезервированы для имен системных объектов, ни одна пользовательская таблица, столбец или другой объект не имеют права обладать именами, начинающимися с этих символов. Формально ничто не мешает создать вам таблицу, название которой начинается с зарезервированных символов, однако документация InterBase настойчиво не рекомендует этого делать.

Возникает вопрос: если данные о структуре таблиц базы данных хранятся в точно таких же таблицах, как и пользовательские, то где хранится информация о самих этих таблицах, которые описывают таблицы? Классический пример проблемы "курицы и яйца" – как одно могло появиться раньше другого, если они взаимозависимы? Решение состоит в том, что системные таблицы в их первозданном состоянии "прошиты" в исходных кодах InterBase и автоматически разворачиваются при создании базы данных в определенном порядке.

Мы уже говорили о таблице RDB$Pages, которая ставит в соответствие физические страницы в файлах базы данных определенным объектам этой базы данных. Структура этой таблицы приведена в табл. 4.24:

Таблица 4.24. Системная таблица RDB$Pages

Column name (имя поля)	Datatype (тип данных)	Description (описание)
RDB$PAGE_NUMBER	INTEGER	Номер физической страницы
RDB$RELATION_ID	SMALLINT	Идентификатор таблицы, для которой распределена страница
RDB$PAGE_SEQUENCE	INTEGER	Порядковый номер этой страницы
RDB$PAGE_TYPE	SMALLINT	Тип страницы – см. таблицу 4.22

Каждая страница данных в базе данных поставлена в соответствие какой-либо таблице, т. е. RELATION. Эту связь поддерживает поле RDB$RELATION_ID, в котором хранится ссылка на таблицу. Как было описано выше, в процессе построения внутреннего образа базы, сервер по определенному жестко "зашитому" в него алгоритму строит эту таблицу и наполняет ее данными. Если быть точным, то в момент построения образа базы данных RDB$Pages является не таблицей, а просто массивом данных определенного формата, известного InterBase. На основании определенного алгоритма сервер считывает данные из этого массива и строит следующую критическую для всей базы таблицу – RDB$Relations. Эта таблица описывает все таблицы базы. Если мы сделаем SQL-запрос:

```
SELECT * from RDB$Relations
```

с целью выяснить, ссылки на какие таблицы содержит RDB$RELATIONS, то увидим, что она содержит и RDB$Pages, и саму себя! Очевидно, что в данном случае сервер слегка "хитрит", "задним числом" подставляя эти и остальные системные таблицы в RDB$Relations, таким образом, "легализуя" их. Сервер реги-

стрирует их как "нормальные" таблицы, в которые можно добавить записи или удалить, т. е. предоставляет стандартный SQL-интерфейс для работы с метаданными.

Может возникнуть вполне резонный вопрос: зачем бы разработчикам InterBase "подстраивать" свои системные данные под пользовательский интерфейс? Ведь внутренние механизмы доступа и чтения были бы быстрее. Разумеется, есть большой резон в том, чтобы предоставить универсальный механизм работы с таблицами, описывающими метаданные.

Дело в том, что логическая структура базы данных состоит не только из таблиц, но и из других объектов. В InterBase существуют следующие объекты:

- Table (таблица);
- View (представление);
- Trigger (триггер);
- Computed_field (вычисляемое поле);
- Validation (проверка);
- Procedure (процедура);
- Expression_index (вычисляемый индекс);
- Exception (исключение);
- User (пользователь);
- Field (поле);
- Index (индекс);
- User-Defined Function (UDF) – функция, определяемая пользователем.

Пока, может быть, не совсем очевидно назначение некоторых из этих объектов, но ясно, что их необходимо описывать и хранить в некотором виде, удобном как для пользователя, так и для доступа из ядра InterBase. Лучше всего это сделать, сохранив все эти объекты в системных таблицах. Их добавление и модификация производятся с помощью SQL-запросов. Не правда ли, элегантное решение? Реализация сервера полностью отделена от конкретной базы данных – все взаимосвязи описываются SQL и его расширениями – языком хранимых процедур и триггеров.

Итак, все объекты сервера хранятся в таблицах. Для каждого вида объектов существует таблица, описывающая все экземпляры, описанные в базе данных. Например, для триггеров есть таблица RDB$Triggers, для процедур – RDB$Procedures, а представления описываются в таблице RDB$Relations.

Рассмотрим подробнее структуру последней таблицы, описывающей все таблицы и представления в базе данных. Структура таблицы RDB$RELATIONS взята из Language Reference for InterBase 6 и приведена в табл. 4.25.

Таблица 4.25. Системная таблица RDB$Relations

Колонка	Тип данных	Длина	Описание
RDB$VIEW_BLR	BLOB	80	BLR: Для представлений (view), содержит BLR (Binary Language

Колонка	Тип данных	Длина	Описание
			Representation) запроса, который InterBase осуществляет каждый раз, когда идет обращение к представлению
RDB$VIEW_SOURCE	BLOB	80	Текст: Для представлений содержит код SQL-запроса, который реализует это представление
RDB$_DESCRIPTION	''	80	Пользовательское описание таблицы или представления
RDB$RELATION_ID	SMALLINT		Содержит внутренний идентификатор таблицы/представления
RDB$SYSTEM_FLAG	''		Определяет тип содержимого таблицы: Пользовательские данные – 0; Системная информация > 0
RDB$DBKEY_LENGTH	''		Длина db$key
RDB$FORMAT	''		Зарезервировано для внутреннего использования InterBase. Содержит счетчик изменений метаданных для данной таблицы
RDB$FIELD_ID	''		Число полей в таблице
RDB$RELATION_NAME	CHAR	31	Уникальное имя таблицы

В описании этой системной таблицы встречается аббревиатура BLR. Чтобы понять, что это такое, придется сделать небольшой экскурс в SQL. Как известно, представления, триггеры и хранимые процедуры – это код, написанный на расширении языка SQL (для каждого сервера СУБД существуют свои расширения). Он приближен к человеческому языку, что позволяет легко составлять на нем запросы. Но InterBase, очевидно, преобразовывает его во что-то более "машинное", а именно в BLR (Binary Language Representation), "двоичное" представление языка (SQL, очевидно). Любой запрос, триггер, представление, хранимая процедура обязательно транслируются в BLR, а уж затем передаются для исполнения ядру InterBase.

BLR

BLR – это специальный язык, используемый в качестве промежуточного звена между SQL-кодом, который пишет программист, и машинным кодом, который "воспринимает" сервер. Никто не пишет непосредственно на BLR – это было бы весьма затруднительно, так как для максимального быстродействия в этом языке используется так называемая обратная польская запись. Вот маленький пример:

```
blr_begin,
         blr_assignment,
            blr_field, 0, 7, 'D','A','T','E','I','Z','M',
            blr_variable, 1,0,
         blr_assignment,
            blr_field, 0, 4, 'R','A','T','E',
            blr_variable, 0,0,
         blr_block,
```

BLR для ваших запросов, процедур, триггеров и других триггеров формируется с помощью специального препроцессора, входящего в состав ядра сервера. Как показано в табл. 4.25, для представлений (VIEW) хранится как их текстовый (исходный) вид, так и скомпилированный вид, т. е. BLR. При обращении к любому объекту, имеющему BLR, сервер выполняет бинарный код объекта, а не интерпретирует каждый раз заново исходный текст этих объектов, что позволяет значительно ускорять выполнение сложных запросов.

Иерархия объектов в InterBase

Чтобы более четко представлять себе, что такое объекты базы данных, мы попробуем построить иерархию объектов базы данных, исходя из принципа "кто кого содержит". Первыми нужно включить в нашу иерархию физические страницы файлов базы данных, как самый низкий уровень организации данных. Затем, очевидно, идут таблицы – основополагающие объекты, которые описывают все остальные типы объектов. Таблицы описывают хранимые процедуры, триггеры, вычисляемые поля, проверки, вычисляемые индексы, исключения и т. д. Обратите внимание: только описывают! Таблицы лишь содержат декларации и определения этих объектов, а сами объекты реализуются через BLR. Поэтому мы можем изобразить таблицы в виде некоторой "рамы", поддерживающей все остальные объекты базы данных. В основание рамы мы положим BLR – как прокладку "реализации", затем поместим "слой" триггеров, хранимых процедур, вычисляемых индексов и представлений.

> Чтобы успокоить специалистов по внутреннему строению InterBase, которые могут возразить, что BLR многих объектов (таких, как представления) хранятся в системных таблицах, заметим, что это отношение довольно трудно изобразить на рисунке, и для простоты мы его опустим. Схема не преследует цель абсолютно точно воссоздать взаимосвязи объектов базы данных, а имеет целью проиллюстрировать их тесную взаимосвязь.

Эти типы объектов объединяет то, что они непосредственно связаны с BLR, который их реализует, без промежуточной логики. Отдельно следует расположить исключения – это специальные виды ошибок, определяемые пользователем. Исключения обрабатываются на уровне ядра InterBase и поэтому не имеют BLR. Такие виды ограничений, как проверки (CHECK), размещаются "поверх" триггеров, поскольку логика ограничений и проверок на самом деле реализуются триггерами.

То, что у нас получилось в результате попытки выстроить иерархию объектов логической и физической структуры базы данных, изображено на рис. 4.11.

Естественно, данная схема лишь приблизительно отражает логическую структуру и взаимосвязи объектов в базе данных и дает только общее представ-

ление о ней. Желающие подробно изучить структуру метаданных базы данных InterBase могут произвести реинжиниринг системных таблиц базы данных и рассмотреть все взаимосвязи между ее объектами, а также обратиться к документации и исходным кодам InterBase. На этой схеме отражены лишь основные объекты базы данных. Давайте кратко опишем основные функции, которые эти объекты выполняют в базе данных.

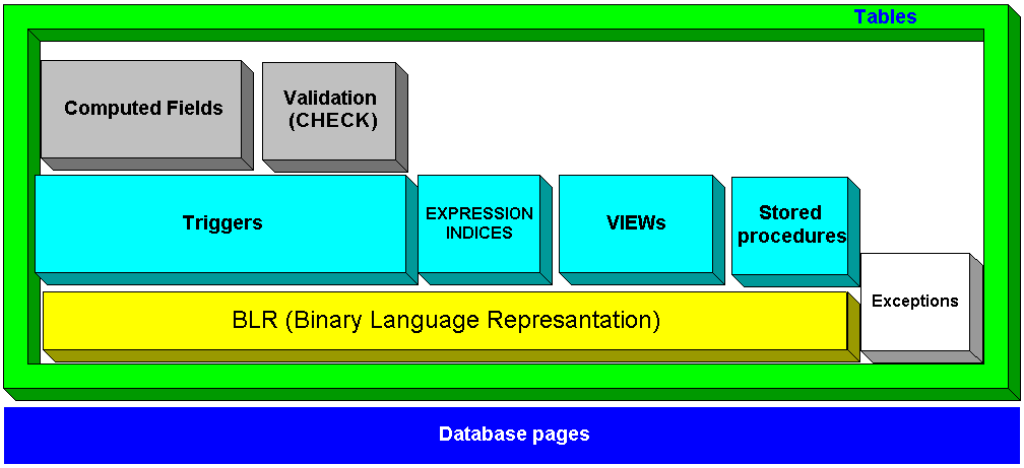


Рис. 4.11. Объекты логической структуры базы данных InterBase

Таблицы (tables) – основной объект, содержащий данные, как пользовательские, так и системные. Таблица имеет уникальное имя и содержит в себе набор поименованных полей. Пользователь может помещать данные, извлекать и модифицировать данные в таблицах. Можно сказать, что таблица аналогична обычным "бумажным" таблицам, начерченным вручную.

Триггеры – исполняемые куски кода, которые применяются для реализации дополнительных действий при операциях с данными. Триггеры исполняются до или после операций вставки, модификации и удаления и позволяют осуществить подстановку значений во вновь создаваемые записи и многое другое.

Хранимые процедуры – мощный инструмент реализации бизнес-логики на уровне базы данных. Выполняясь на уровне сервера, хранимые процедуры работают очень быстро и позволяют совершать множество операций над наборами данных. Хранимые процедуры InterBase возвращают стандартные SQL-наборы данных, над которыми можно производить все SQL-операции, включая объединение с другими таблицами.

Представления (VIEW) – это скомпилированные SQL-запросы, выполняющиеся на сервере. Представления позволяют гибко организовывать наборы данных, переносить часть бизнес-логики на сервер.

Проверки (Validation) – ограничения, накладываемые на значения полей в таблице. Например, можно указать, что данное поле будет принимать только положительные значения. Ограничения на значения полей реализуются с помощью триггеров и позволяют эффективно управлять ссылочной целостностью на

уровне базы данных. Обычно ограничения применяются для того, чтобы предотвратить помещение неправильных значений в таблицу.

Пользователи (Users) – InterBase позволяет завести для работы с базой данных несколько пользователей и распределять между ними права доступа к различным объектам базы данных. Таким образом, можно гибко управлять разрешениями на те или иные операции с базой данных.

User-Defined Functions (UDF) – функции, определяемые пользователем. Это одна из наиболее мощных возможностей InterBase, позволяющая расширять стандартный SQL-интерфейс своими собственными функциями. Например, функции работы со строками, такие, как UPPER (привести все символы к верхнему регистру), реализованы в стандартной UDF-библиотеке, поставляющейся в комплекте с InterBase. Свойство создавать собственные UDF дает разработчикам возможность расширять функциональность InterBase практически любыми функциями. Для создания UDF подходит любая среда программирования, которая позволяет производить динамические библиотеки (Visual C++, C++ Builder, Delphi и т. д.).

Заключение

В этой главе впервые были рассмотрены вопросы реализации хранения и обработки данных внутри базы данных InterBase. К сожалению, даже провести краткий обзор данной темы не удается, не прибегая к большому числу терминов и к большому количеству неточных аналогий. При более подробном изложении физической и логической структуры базы данных в любом случае пришлось бы обращаться к исходным кодам InterBase, но это потребовало бы уже совершенно отдельной книги.

Тем не менее мы считаем, что каждому программисту было бы полезно ознакомиться с "начинкой" того продукта, который он ежедневно использует. Возможно, кто-то из читателей пойдет дальше – попробует разобраться в исходном коде InterBase/Firebird и примкнет к команде разработчиков Firebird, внеся туда множество новых интересных идей.

Обзор современных версий семейства InterBase

Yaffil – российский клон СУБД InterBase

Введение

Эта глава посвящена СУБД Yaffil (англ. – дятел) и его особенностям.

После открытия исходного кода InterBase разработчики, использующие сервер в своих проектах, стали пытаться его усовершенствовать, приспосабливая к своим нуждам и улучшая характеристики. В то же время ясно, что самостоятельная модификация кода разрозненными разработчиками исключительно в собственных целях приведет к появлению несовместимых версий и трудностям в сопровождении. В мире программного обеспечения с открытым исходным кодом (Open Source) подобная неприятная ситуация достаточно распространена и имеет название code forking.

В конце 2001 года в результате объединения усилий группы российских разработчиков, использующих InterBase на Windows NT, на свет появился проект Yaffil. За основу разработчики взяли исходный код сервера Firebird 1.0, поскольку он является динамично развивающимся проектом с открытым исходным кодом.

Почему было принято решение создавать новый клон, вместо того чтобы интегрировать изменения с проектом Firebird?

К сожалению, политика координаторов и участников проекта Firebird в то время являлась достаточно жесткой, особенно это касалось внедрения новых возможностей. Главным приоритетом команды Firebird было и остается создание стабильной версии на базе существующего исходного кода при сохранении полной платформенной независимости.

В то же время разработчики Yaffil считали очень перспективным направлением интеграцию сервера с Windows NT, что требовало введения специфичных для данной операционной системы возможностей в сервер, что было неприемлемо для проекта Firebird 1.0.

В 2004 году многое изменилось и в команде Firebird, и в команде Yaffil. Стало ясно, что объединение усилий двух команд гораздо эффективнее, чем работа порознь. Было решено, что над версией Firebird 2.0 будет вестись совместная работа. До выпуска 2.0 в Yaffil будут исправляться ошибки, однако вся новая функциональность будет переноситься в Firebird 2.0, который будет совместим с Yaffil 1.0

Приоритетные направления развития Yaffil

Интеграция с платформой Windows NT

Изначально InterBase разрабатывался на платформах Unix и только в начале 90-х годов в версии 4.0 был перенесен на Windows NT. К сожалению, при переносе кода мало внимания было уделено платформозависимой оптимизации под Windows NT, в результате данный вариант сервера оказался менее эффективным, чем мог бы быть. В то же время число установок InterBase на Windows NT составляет значительную (а возможно, и большую) часть, поэтому такая ситуация является достаточно печальной.

Кроме того, Windows NT обладает рядом возможностей, использование которых в сервере СУБД позволит добиться нового уровня функциональности и увеличить отдачу при использовании сервера в прикладных системах. Среди таких возможностей назовем интерфейс для аутентификации пользователей по протоколам NTLM или Kerberos, при использовании которого нет необходимости вводить имя пользователя и пароль при каждом соединении, встроенные криптографические средства, координатор распределенных транзакций (MS DTC), асинхронный ввод-вывод, интерфейс WinSock2 и другие. Все перечисленные возможности широко используются другими серверами баз данных, однако полностью игнорируются сервером InterBase/Firebird.

Производительность

Производительность систем на базе СУБД является одним из ключевых факторов при выборе сервера. Учитывая это, при разработке сервера Yaffil большое внимание уделяется оптимизации исходного кода, используются средства анализа и профайлинга производительности.

Надежность и безопасность

В сервере Yaffil исправлено большое число критических ошибок, многие из которых приводят к порче данных. В предыдущих версиях InterBase такие опасные ситуации, как порча памяти (вследствие ошибки в коде сервера, некорректной UDF или испорченных метаданных БД), протекали внешне незаметно до тех пор, пока база данных не была непоправимо испорчена.

Ошибка при работе с памятью в Yaffil, как правило, приводит к немедленному останову сервера с потерей только последних обновляющих транзакций.

В Yaffil, как и в InterBase 7.0, исправлена дыра в безопасности, связанная с использованием внешних таблиц для получения доступа к любому файлу системы. Также в Yaffil исправлена серьезная ошибка, связанная с обработкой структурированных исключений (SEH) Win32 API

Отличительные особенности сервера Yaffil

Улучшенная производительность

Производительность является одним из ключевых факторов, определяющих пригодность сервера СУБД для использования в конкретном приложении. Производительность определяет максимальную нагрузку, которую сервер может нести на выбранной аппаратной платформе, выраженную, например, в количестве одновременно работающих пользователей или количестве операций, выполняемых в единицу времени. К сожалению, скоростные характеристики линейки серверов InterBase часто уступают показателям конкурентов, таких, как MSSQL или Sybase SQL Anywhere. По производительности Yaffil далеко опережает своих аналогов из клона InterBase. При выполнении тех же самых запросов Yaffil работает в 2–3 раза быстрее, чем InterBase 6.0 и Firebird 1.0. Такие результаты достигнуты благодаря многочисленным улучшениям алгоритмов и оптимизацией кода, большое число критических алгоритмов переписаны на ассемблере x86, убраны ненужные системные вызовы.

Примечание: В настоящее время (к моменту выхода 3-го издания) разработчики как Borland InterBase, так и Firebird исправили множество ошибок оптимизатора и в последних версиях значительно улучшили производительность.

Улучшенный оптимизатор запросов

Оптимизатор – своего рода "мозг" сервера, и степень его интеллектуальности может кардинально повлиять на скорость работы приложений. Неверно выбранный план может привести к увеличению времени выполнения запроса в тысячи раз. За время своей эволюции от версии InterBase 4.0, не способной вообще оптимизировать явные соединения, и до последних версий в оптимизатор InterBase разработчиками Borland вносились изменения, которые хотя и приводили к улучшению планов для определенных случаев, но часто непредсказуемо сказывались на других запросах. Например, в пятой версии InterBase научился корректно оптимизировать соединения в явном синтаксисе ANSI SQL, но другой способ выборки индексов в ряде случаев приводил к катастрофическому увеличению времени выполнения запросов. Подобные неприятности появились в версии 6.0, в результате многие разработчики не могли перенести свои системы на современную платформу.

Проанализировав большое число проблемных случаев с оптимизацией InterBase версий 4.x, 5.x и 6.x, разработчики Yaffil внесли ряд улучшений, в результате чего в большинстве случаев можно вообще отказаться от применения ручных планов. При этом при переходе на Yaffil со старых версий InterBase случаев ухудшения автоматических планов практически не наблюдается.

Однако в тех случаях, когда ручного планирования не избежать, Yaffil предоставляет больше возможностей для этого.

Перечислим некоторые улучшения оптимизатора Yaffil.

Выбор индекса с меньшим числом полей при наличии индекса с большим числом полей

Этот случай часто возникает, когда в таблице имеется большое количество индексов, в том числе составных. Оптимизатор Interbase/Firebird всегда пытался выбрать индекс, который охватывает множество полей, в то время как существовал более быстрый и компактный индекс. Оптимизатор Yaffil автоматически разрешает данную ситуацию. Вот пример:

Таблица из трех полей Table1 (F1, F2, F3)

На нее созданы три индекса: IDX_F1(F1), IDX_F1_F2(F1,F2), IDX_F1_F2_F3(F1,F2,F3)

Выполняем запрос:

```
select * from Table1 where F1 = параметр
```

Получаем следующие планы для использования индексов:

Interbase 6.5/FireBird 1.0:

```
PLAN (T INDEX (IDX_F1_F2_F3))
```

Yaffil:

```
PLAN (T INDEX (IDX_F1))
```

Interbase 6.5/FireBird 1.0 используют индекс с максимальным количеством полей, хотя очевидно, что в данном случае надо использовать единственный индекс по полю F1, как это и делает Yaffil. Это очень распространенная ошибка, приводящая к замедлению выполнения запроса, обойти которую без явного планирования достаточно сложно.

Возможность использовать индекс с начальными сегментами, удовлетворяющими условию и сортировкой по остальным

Этот случай возникает, когда выборка делается по одному полю, а сортировка – по другому, при этом имеется индекс по обоим полям. Пример: пусть существует таблица из трех полей T(F1,F2,F3), есть индексы на следующие сочетания полей: на одно поле IDX_F1(F1), на два первых поля IDX_F1_F2(F1,F2), на все три поля IDX_F1_F2_F3(F1,F2,F3).

Производим запрос следующего вида:

```
 select * from T where F1=.. order by F2
```

Получаем планы следующего вида:

FireBird/InterBase 6.5:

```
PLAN SORT(T INDEX (IDX_F1_F2_F3))
```

Yaffil:

```
PLAN (T ORDER (IDX_F1))
```

Исключение из обработки индексов с сильно различающейся селективностью

Пример: пусть у нас есть таблица из трех полей T(F1,F2,F3) с индексами на каждое поле IDX_F1(F1), IDX_F2(F2) и IDX_F3(F1). Пусть здесь F1 является уникальным полем (первичный ключ, например), а F2 – неуникальным. Производим запрос:

```
select * from T where F1 = параметр1 and F2 = параметр2 ...
```

В результате получаем следующие планы:

FireBird/InterBase 6.5:

```
PLAN (T INDEX (IDX_F1, IDX_F2))
```

Yaffil:

```
PLAN (T INDEX (IDX_F1))
```

Оптимизатор Yaffil всегда объединяет индексы с учетом селективности.

Отметим, что в качестве критериев отбора индексов используется не только селективность, но и категории условий, используемых в ограничениях. Так, например, "вес" операции "=" больше, чем "вес" операций "<" и ">".

Оптимизация сетевого трафика

Как правило, сервер СУБД устанавливается не на одном компьютере вместе с клиентом, а используется через локальную сеть. При этом на времени отклика сервера сказываются задержки при передаче данных по сети, независимо от того, насколько мощный сервер установлен.

Объем сетевого трафика Yaffil значительно уменьшен за счет более эффективной передачи полей типа VARCHAR, так как передаются данные фактической длины, а не объявленной в описании типа. Аналогичное улучшение есть в Borland InterBase, начиная с версии 6.5.

Эффективная работа с временными файлами сортировки

В сервере InterBase сортировка всегда выполняется с использованием временных файлов, независимо от количества доступной памяти. Операции чтения/записи временных файлов дополнительно нагружают дисковую подсистему, что может отрицательно сказаться на скорости работы сервера, особенно при наличии параллельно работающих соединений интенсивно обращающихся к диску. Поэтому сервер Yaffil открывает временные файлы сортировки с флагом FILE_ATTRIBUTE_TEMPORARY, что предотвращает сброс кэш-буферов на диск операционной системой при наличии достаточного объема кэш-памяти.

При наличии большого количества оперативной памяти для сервера Windows NT, на котором выполняется Yaffil, имеет смысл увеличить приоритет файлового кэша на использование памяти. При этом менеджер памяти будет стараться использовать свободную физическую память как файловый кэш вместо предоставления ее работающим приложениям. За данный параметр настройки отвечает значение LargeSystemCache ключа системного реестра

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session
Manager\Memory Management.
```

Для сервера InterBase рекомендуется выставить это значение в 1. Для того чтобы изменение вступило в силу, необходимо перезагрузить систему.

Обратите внимание, что некоторые серверные приложения, такие, как Microsoft SQL Server, могут переустанавливать это значение в 0 при своей инсталляции, поскольку не используют файловый кэш.

Оптимальная структура хранения записей

InterBase использует эффективный способ хранения записей на страницах базы данных, используя алгоритм RLE (run length encoding – кодирование последовательностей) при размещении данных, за счет которого базы данных InterBase являются компактными. Несмотря на то что алгоритм является очень простым, он тем не менее требует вычислительных ресурсов при записи и чтении данных на странице.

В зависимости от характера данных упаковка может оказаться нецелесообразной. Наибольший эффект достигается при сжатии "хвостов" из пробелов длинных текстовых строк (типы данных CHAR и VARCHAR), в то время как короткие нетекстовые данные практически несжимаемы.

Для управления сжатием в Yaffil введен параметр конфигурации SQZ_BLOCK. Этот параметр определяет минимальный размер блока строки таблицы (в байтах), начиная с которого будет производиться сжатие. Блоки строки таблиц менее указанного размера хранятся в исходном виде. С увеличением параметра объем базы данных возрастает за счет сокращения затрат времени процессора на упаковку. В то же время увеличение количества данных больше нагружает дисковую подсистему. В зависимости от характера данных (главным образом количества текстовых полей в таблицах) и определенных параметров процессора и дисков системы существует некоторое оптимальное значение SQZ_BLOCK, обеспечивающее максимальную производительность. Алгоритм упаковки, реализованный в Yaffil, дополнительно выравнивает начало и размер сжатого блока на границу машинного слова.

Ускоренная работа с индексами

Гораздо быстрее работают операции поиска по индексам (часто используемые в соединениях), а также построения индексов. Это достигнуто за счет тщательной оптимизации кода индексирования.

Улучшенная стратегия вычисления предиката IN и условий, объединенных по OR

По сравнению с InterBase и другими клонами, Yaffil выполняет предикат IN гораздо более эффективно. При этом используется только одно построение битовой карты для индекса, независимо от числа параметров. В других клонах InterBase получим значительное замедление при большом числе параметров, (особенно если IN не ограничивает основной объем выборки), несмотря на одинаковый план исполнения запроса. То же самое произойдет и для условия вида OR:

```
select * from T where F1 = .. or F2 = ... or F2
```

Ускоренное обновление данных

Наибольший выигрыш в скорости при использовании Yaffil наблюдается при обновлении данных.

Переработаны алгоритмы, связанные с управлением деревом "грязных страниц" буферного кэша, в результате чего устранено замедление при массивных

обновлениях данных в рамках одной транзакции. Ликвидирована известная проблема InterBase, при которой задание числа буферов больше некоторого предела (около 10 000) приводит не к увеличению скорости, а совсем наоборот, при этом возможны даже зависания (известная проблема "10 000 буферов").

Устранена деградация скорости при обновлении большого числа рядов данных, а также при откате большого числа изменений. Быстрее выполняется массовая вставка данных также за счет оптимизации кода.

Уменьшение времени, необходимого для резервного копирования и восстановления

В InterBase отсутствует возможность инкрементального резервного копирования, при этом сам процесс backup/restore из-за архитектурных особенностей проходит медленнее по сравнению с серверами СУБД, где запись копии происходит постранично. Резервное копирование большой базы может занимать несколько часов, при этом хоть и возможна работа пользователей, но время отклика ухудшается.

На Yaffil резервное копирование и восстановление происходит в несколько раз быстрее, особенно заметно это на больших базх данных (отсутствует деградация скорости копирования со временем).

Индексы по выражениям

Индексы по выражениям (Expression Indexes) используются в тех случаях, когда необходимо обеспечить быстрый поиск или сортировку по значениям, вычисляемым на основе полей таблицы. Необходимый индекс определяется следующим образом:

```
CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]]
INDEX index ON table COMPUTED BY (expression)
```

где index – имя индекса, expression – выражение, построенное на основе полей таблицы. В выражении можно использовать арифметические операции, встроенные функции и UDF. Во время оптимизации запросов, если выражение совпадает с выражением, указанным в условии WHERE или в условии соединения, будет использован индекс.

Например, необходимо сделать выборку записей, имеющих поле типа дата, для определенного месяца по всем годам (на примере базы данных Employee из комплекта поставки InterBase 6):

```
CREATE INDEX employee_hire_date_month_idx ON employee COMPUTED
BY (EXTRACT(MONTH FROM hire_date));
```

Попробуем выполнить запрос и посмотрим на выбранный оптимизатором план:

```
SELECT * FROM employee
WHERE EXTRACT(MONTH FROM hire_date) = 1
PLAN (EMPLOYEE INDEX (EMPLOYEE_HIRE_DATE_MONTH_IDX))
```

Вычисляемое выражение должно полностью определяться значениями полей таблицы, индекс на основе, например, функции CURRENT_TIMESTAMP, скорее всего, будет бесполезен, хотя Yaffil не запрещает использование подобных

выражений. В вычисляемых выражениях нельзя использовать подзапросы. Очень интересные возможности появляются при использовании User-Defined Functions (UDF) в вычисляемых выражениях, с их помощью можно выполнять эффективный поиск по практически любому условию.

Уменьшение размера, занимаемого индексами

Индексы, построенные по текстовым полям с национальным порядком сортировки, занимают в среднем на одну треть меньше места на диске по сравнению с Interbase.

Выражения в значениях по умолчанию для доменов

Yaffil позволяет использовать сложные выражения для значений по умолчанию в доменах:

```
CREATE DOMAIN NEW_DOMAIN AS INTEGER DEFAULT
(GEN_ID(NEW_GENERATOR, 1))
```

Подобная возможность позволяет не писать дополнительные триггера для выполнения схожих действий.

Удобная операция объединения строк

Yaffil позволяет объединять строки с использованием оператора "||" при превышении размера результата максимальной длины строки. При этом результирующая строка будет иметь максимально допустимый сервером размер, а ошибка переполнения возникает, только если действительные данные пользователя не могут быть размещены в результирующей строке. Такое поведение оператора "||" соответствует привычному поведению обычных арифметических операторов над числами.

В качестве примера приведем вариант с объединением двух полей:

```
CREATE TABLE T(V1 VARCHAR(20000), V2 VARCHAR(20000)).
```

При попытке написать V1 || V2 Interbase/Firebird выдадут ошибку переполнения еще на этапе компиляции. Сервер Yaffil в качестве результата сформирует строку VARCHAR(32765). Ошибка переполнения возникнет, только если количество символов объединения V1 и V2, исключая концевые пробелы, превысит 32765.

Расширенные возможности указания пользовательских планов

Не всегда встроенный оптимизатор может выбрать оптимальный план. Причиной этого может быть отсутствие подробной статистики по индексам и полям, без которой трудно оценить стоимость выполнения варианта или слишком большое число вариантов соединений. В таких случаях приходится применять явные планы в запросах.

Yaffil расширяет возможности использования явных планов, тем самым предоставляя возможность дальнейшего ускорения работы приложений.

В Yaffil появилась возможность указывать явные планы в некурсорных операторах обновления данных, таких, как UPDATE и DELETE. Дополнительно этим можно добиться обновления или удаления строк данных в заданном порядке, указав использование индекса в условии ORDER.

Как известно, в InterBase не разрешается использование явных планов в тексте триггеров. Yaffil снимает это ограничение.

Имена индексов ограничений

Использование явных планов в Yaffil в триггерах и процедурах существенно упрощается благодаря возможности именования индексов, автоматически создаваемых сервером для ограничений первичных, внешних ключей и ограничений уникальности. В версиях InterBase такие индексы приобретают системные имена в формате RDB$PRIMARYX для индексов первичных ключей, RDB$FOREIGNX для индексов внешних ключей и RDB$X для индексов ограничений уникальности. X обозначает номер индекса данного типа по порядку с момента создания базы данных (или восстановления из резервной копии). Так как эти номера могут измениться при следующем восстановлении базы данных, фиксировать такие индексы в плане становится опасным. Возникает тупиковая ситуация: автоматический план неэффективен, явный план записать нельзя. Разработчикам приходится создавать собственные индексы, целиком дублирующие системные. Однако появление новых индексов на таблице влечет к увеличению дискового пространства, занимаемого базой, замедлению операций обновления, кроме того повышению вероятности выбора оптимизатором неверного плана из-за увеличения числа вариантов соединений таблиц.

В Yaffil индексы для ограничений могут принимать имена соответствующих ограничений.

Нужное поведение включается параметром CONSTRAINT_INDEX_NAME в конфигурационном файле. Например:

```
SQL> create table T (id into not null, constraint PK_T primary
key (id));
SQL> show index;
PK_T UNIQUE INDEX ON T(ID)
```

Улучшенное время отклика для версии SuperServer

В серверах Bolrand InterBase версии ниже 7.0, использующих архитектуру SuperServer, одновременное обслуживание нескольких клиентов реализовано по схеме многопоточного сервера. Однако переключение процессора между потоками (диспетчеризация) происходит не по требованию операционной системы, а в моменты времени, выбираемые активным потоком "добровольно". Такая схема очень похожа на реализацию многозадачности в Windows 3.1 и называется *невытесняющей многозадачностью*.

С каждым потоком связано числовое значение, первоначально равное величине кванта времени. При прохождении потока через определенные точки в коде это значение уменьшается на единицу, пока не достигнет нуля. В этот момент квант времени считается исчерпанным и активный поток передает управление другому. Ясно, что обеспечить равный доступ каждого потока к процессору при

таком подходе невозможно. На практике эта проблема проявляется в резком увеличении времени выполнения оперативных запросов при одновременном выполнении длительных, "тяжелых" запросов.

В сервере Yaffil данная проблема ослаблена за счет введения дополнительных точек переключения в наиболее часто повторяющихся участках кода сервера. Для рабочих потоков сервера и отдельно для потока сборки мусора есть возможность задавать величину кванта времени с помощью параметров конфигурации THREAD_QUANTUM и SWEEP_THREAD_QUANTUM. Параметр FORCE_RESHEDULE активизирует дополнительные точки переключения. В результате распределение процессорного времени между рабочими потоками происходит более равномерно.

Улучшенный протокол локальных соединений (XNET)

Локальное соединение в InterBase выполняется с использованием буферов разделяемой памяти и обеспечивает более высокую производительность по сравнению с другими протоколами (TCP и Named Pipes). В документации это называется локальным протоколом или IP Server (IPS) по внутренней терминологии Borland. Вместе с тем локальное соединение в InterBase обладает двумя серьезными недостатками, часто делающими его использование невозможным. Прежде всего, локальное соединение не допускает одновременной параллельной (многопоточной) работы нескольких соединений. Второе клиентское соединение будет заблокировано до момента завершения первого.

Кроме того, для установления соединения используется оконное сообщение, посылаемое клиентом специальному скрытому окну сервера. Если сервер InterBase работает как служба NT и клиентское приложение также работает как служба, они могут использовать разные desktops, при этом посылка оконных сообщений между ними невозможна.

Типичный пример подобной конфигурации – WEB-сервер Microsoft Internet Information Server (IIS), выполняющий приложения, обращающиеся к базам данных InterBase. Традиционно в таких случаях рекомендуется использовать протокол TCP для установления соединений из сервисов NT, что отрицательно сказывается на производительности.

XNET также является реализацией локального соединения с использованием буферов разделяемой памяти, поэтому его скорость передачи данных идентична "старому" локальному протоколу (IPS). Отличие состоит в том, что его реализация допускает одновременное использование на разных потоках без взаимного блокирования. Для установления соединения больше не используются окна и оконные сообщения, поэтому соединения надежно устанавливаются из служб Windows NT.

Оригинально XNET был разработан специалистами фирмы Borland и планировался как замена ненадежному старому локальному протоколу. Начало разработки датировано в исходном коде 1995 годом.

В сервере Yaffil по сравнению с оригинальным кодом внесены изменения, направленные на повышение надежности работы в случае неожиданного "падения" одной из сторон. Возможность соединения с сервером из служб NT также впервые была реализована в Yaffil.

Строка соединения по XNET аналогична обычной строке локального соединения. Протокол XNET не совместим по локальному подключению с другими версиями InterBase/Firebird, поэтому необходимо использовать клиентскую биб-

лиотеку (GDS32.DLL), соответствующую версии сервера. В текущей версии XNET недоступен в Yaffil Classic Server.

Ограничение времени ожидания для транзакций (Lock timeout)

При возникновении конфликта обновления записи в InterBase возможны два варианта поведения транзакции, задаваемых параметром WAIT (isc_tpb_wait / isc_tpb_no_wait), – бесконечное ожидание разрешения конфликта или немедленная выдача ошибки. Режим с ожиданием часто удобнее, так как нет необходимости повторять операцию в случае конфликта, но такой режим является очень опасным из-за возможности бесконечной блокировки приложения.

В сервере Yaffil добавлена возможность ограничивать время ожидания разрешения конфликта заданным интервалом времени. Для этого служит параметр конфигурации LOCK_TIMEOUT, задающий время в секундах. Положительное значение от 1 до 32 767 определяет время ожидания WAIT транзакций. Отрицательное число определяет бесконечное время ожидания. Нулевое значение эффективно превращает WAIT транзакции в NOWAIT. Значение по умолчанию –1 (минус один), что обеспечивает совместимость с другими версиями. Параметр не оказывает влияния на транзакции, запущенные в режиме NOWAIT.

В следующих версиях планируется ввести константу блока TPB (transaction parameter block), управляющую временем ожидания при запуске каждой транзакции индивидуально.

Расширения SQL

В сервере Yaffil реализовано несколько дополнительных языковых конструкций SQL по сравнению с Interbase/Firebird:

Инструкция IIF

Инструкция IIF позволяет реализовать дополнительную логику в запросах.

Синтаксис:

```
IIF '(' search_condition ',' value_if_true ',' value_if_false
')'.
```

Выполняя инструкцию IIF, сервер вычисляет выражение search_condition. Если search_condition, то результатом IIF является выражение value_if_true, в противном случае value_if_false.

Пример:

```
select iif(rc.rdb$collation_id = 0, 'ДА', 'НЕТ') from
rdb$collations rc
where rc.rdb$collation_name = 'WIN1251'
```

Выполнив запрос, получим – "ДА".

Инструкцию IIF можно применять и при вычислении выражений.

Пример:

```
a = b + iif(c is null, 0, c);
```

Инструкция INSERT INTO … FROM … UNION …

Сервер Yaffil, в отличие от InterBase/Firebird, позволяет использовать объединения UNION для формирования данных на вставку.

Пример:

```
insert into t_a (id) select b.id from b union select c.id from c
```

Выражения в EXCEPTION

Сервер Yaffil расширяет синтаксис инструкции exception. Допускаются три варианта использования:

1. exception 'исключение'; Этот вариант соответствует синтаксису Interbase/Firebird – сервер выбрасывает исключение, заданное соответствующим идентификатором.
2. exception 'исключение' 'выражение'; В этом случае сервер также выбрасывает исключение, но его текст заменяется на результат вычисления выражения.
3. exception; Сервер выбрасывает последнее сформированное исключение. Если до выполнения этой инструкции исключений выброшено не было, то инструкция игнорируется. Этот вариант используется для повторного выбрасывания исключения в обработчиках ошибок WHEN.

Системные переменные ROWS_AFFECTED, GDSCODE, SQLCODE, TRANSACTION_ID, CONNECTION_ID

- Переменная ROWS_AFFECTED содержит количество записей, модифицированных в результате выполнения последнего запроса.
- Переменная GDSCODE содержит значения инструкции gdscode в обработчике WHEN, однако может использоваться вне контекста WHEN.
- Переменная SQLCODE содержит значения инструкции sqlcode в обработчике WHEN, однако может использоваться вне контекста WHEN.
- Переменная TRANSACTION_ID содержит номер транзакции.
- Переменная CONNECTION_ID содержит номер подключения.

Группировка по номеру столбца

Сервер Yaffil расширяет синтаксис инструкции group by. Допускается указывать номера столбцов для группировки, как в инструкции oreder by.

Пример:

```
select count(a), b from t group by 2
```

Значения переменных по умолчанию

Сервер Yaffil расширяет синтаксис инструкции declare variable. Можно не указывать ключевое слово variable и указать инициализирующее значение переменной.

Пример:

```
declare k = 0;
```

Тип данных BIGINT

Дополнительный тип данных BIGINT является аналогом типа данных NUMERIC (18,0) и предлагает более лаконичное и понятное название для 64-битного целого.

Дополнительные национальные кодовые страницы и порядки сортировки

В Yaffil добавлены следующие национальные кодовые страницы и порядки сортировки:

- CS_WIN1257 – страны Прибалтики, кодовая страница Windows – 1257

Порядки сортировки:

- WIN1257_EE – эстония
- WIN1257_LV – литва
- WIN1257_LT – латвия
- CS_KOI8R – Россия KOI8, кодовая страница Windows – 20866

Порядки сортировки:

- KOI8R
- KOI8R_RU
- CS_KOI8U – Украина KOI8-U, кодовая страница Windows – 21866

Порядки сортировки:

- KOI8U
- KOI8U_UA

Также для кодовой страницы WIN1251 добавлены порядки сортировки:

- WIN1251_UA
- WIN1251_RU

Группировка по встроенным функциям и UDF

Разрешена группировка и использование встроенных функций и UDF.
Пример:

```
select sum(vent) from sales group by extract(year from sale
date)
```

Ограничение результатов выборки FIRST/SKIP

Как и в Firebird 1.0, результат выборки SELECT может быть ограничен с использованием инструкций FIRST/SKIP. (В Borland InterBase начиная с версии 6.5 используется аналогичная по назначению конструкция ROWS.)

При значении аргумента FIRST, равном 0, результатом выборки будет пустое множество, в отличие от Firebird 1.0, где будет возбуждено исключение с сообщением о неверном параметре.

Увеличение глубины рекурсии процедур и триггеров

Количество рекурсивных вызовов процедур и триггеров увеличено до 1000.

Использование переменной окружения ISC_PATH

В Yaffil расширены возможности использования переменной окружения ISC_PATH для задания префикса к пути базы данных. Переменная ISC_PATH используется, если в имени базы данных, указываемом клиентом, не содержит каталогов или имени сервера. Переменная ISC_PATH может использоваться как на клиенте, так и на сервере. Примеры:

1. Использование ISC_PATH на сервере. Пусть базы данных на сервере находятся в каталоге c:\database. Определим переменную ISC_PATH=c:\database. Далее можно использовать на клиентском компьютере строку соединения вида servername:database.gdb для открытия базы данных c:\database\database.gdb.
2. Использование ISC_PATH на клиенте. Пусть базы данных располагаются на сервере servername в каталоге c:\database. Определим на клиенте переменную ISC_PATH как servername:c:\database. После этого клиент сможет обращаться к базам данных только по короткому имени файла БД, например CONNECT sales.gdb. Используем внешний файл в базе sales.gdb:

```
CREATE TABLE customers EXTERNAL FILE
"sales_files/customers.txt" ( ... );
```

Безопасная работа с внешними таблицами

Файлы внешних таблиц могут располагаться ТОЛЬКО в одном из каталогов, разрешенных конфигурационным параметром EXTERNAL_FILE_DIRECTORY, а также в их подкаталогах.

По умолчанию конфигурационный файл не содержит параметров EXTERNAL_FILE_DIRECTORY, поэтому использовать внешние файлы вообще не разрешается. Если нужно полностью снять ограничения на размещение внешних файлов, следует задать корневые каталоги дисков в EXTERNAL_FILE_DIRECTORY, например:

```
EXTERNAL_FILE_DIRECTORY "c:\"
EXTERNAL_FILE_DIRECTORY "d:\"
```

Таким образом, любой файл на диске становится доступным для доступа через внешние таблицы, что соответствует прежнему, небезопасному поведению InterBase pre-7 и Firebird 1.0. Из-за серьезных проблем с безопасностью НЕ РЕКОМЕНДУЕТСЯ использовать такие установки.

Каждому каталогу, сконфигурированному для использования в качестве хранилища внешних таблиц, может быть присвоено логическое имя.

Логическое имя каталога необязательно и задается вторым аргументом:

```
EXTERNAL_FILE_DIRECTORY "c:\databases\files" myfiles
```

На логическое имя можно ссылаться при задании внешней таблицы. Для этого нужно указать логическоеq имя каталога после символа $ в начале имени внешнего файла. Например:

```
CREATE TABLE customers EXTERNAL FILE "$myfiles/customers.txt" ( ... );
```

Другой способ задания внешних таблиц состоит в указании абсолютного пути к файлу. При этом файл также может располагаться только в разрешенных каталогах. Например,

```
CREATE TABLE customers EXTERNAL FILE
"c:/databases/files/customers.txt" (..);
```

И наконец, можно задавать файл внешней таблицы с указанием имени относительно каталога базы данных:

```
CREATE TABLE customers EXTERNAL FILE "files/customers.txt" ( ... );
```

При этом каталог, в котором находится файл, должен быть разрешен параметром EXTERNAL_FILE_DIRECTORY.

НЕ РЕКОМЕНДУЕТСЯ разрешать для размещения файлов внешних таблиц каталоги с базами данных, поскольку в этом случае пользователи получают доступ к файлам этих баз данных. Если есть желание разместить внешние таблицы рядом с базой данных, то следует создать подкаталог для внешних файлов для каждой базы данных и разрешить его в EXTERNAL_FILE_DIRECTORY. Допустим, все базы данных лежат в c:\databases, среди которых sales.gdb. Создаем каталог sales_files, разрешаем его для использования:

```
EXTERNAL_FILE_DIRECTORY "c:\databases\sales_files"
```

Классическая архитектура на Windows NT (Yaffil CS)

Реализация классической архитектуры Yaffil CS на платформе Windows NT является значительным преимуществом сервера Yaffil по сравнению с другими вариантами InterBase/Firebird, существующими на сегодняшний день. Классическая ветвь InterBase для Windows NT прекратила развитие в 1994 году в связи с началом разработки варианта SuperServer для версии 4.0. Предполагалось, что SuperServer быстро заменит классическую архитектуру, однако реализация SuperServer, имеющая приемлемые характеристики при использовании с большой нагрузкой, не была удачной. В связи с этим до недавнего времени версии InterBase для платформ Solaris и Linux поставлялись в двух вариантах – SuperServer и Classic Server.

Классическая архитектура обладает следующими преимуществами:

- Равномерное распределение нагрузки между выполняющимися запросами на разных соединениях.
- Эффективное использование многопроцессорных систем (SMP).
- Более полное использование оперативной памяти.
- Встраиваемый (embedded или in-process) сервер.

Другими словами, сферой использования Classic Server являются высокопроизводительные системы, обслуживающие одновременно большое число подключений, в то время как SuperServer более эффективен на системах небольшого размера.

Не будем здесь останавливаться на отличиях архитектур Classic и SuperServer, подробно описанных выше в этой книге, рассмотрим лишь особенности реализации Yaffil Classic Server.

- В отличие от сервера Interbase 4.0 для Windows NT, Yaffil CS способен запускаться не только как служба Windows NT, но и как приложение. Во втором случае в области system tray панели задач Windows появляется красная иконка Yaffil Server. В строке версии вместо пары букв WI (платформа Windows Intel) используются буквы NI (платформа NT Intel), поскольку именно так обозначала себя первая версия InterBase CS 4.0 для Windows NT.
- Количество соединений, поддерживаемых Yaffil Classic Server, ограничено только ресурсами системы, в основном оперативной памятью. Ограничение для InterBase CS 4.0 в 90–120 соединений при запуске сервиса ibremote с параметром "not allowed to interact with desktop" снято в Yaffil CS.

Встраиваемый сервер

Как известно, первые версии сервера InterBase, работающие под операционной системой UNIX и другими, более экзотическими ОС, использовали прямое связывание кода сервера с клиентским приложением. Строго говоря, термины клиент и сервер здесь не очень уместны, так как существует всего один процесс.

Можно провести аналогию с технологией COM (Component Object Model), в которой используется термин "in-process server" (сервер внутри процесса) для обозначения компонентов, загружаемых в адресное пространство клиентского приложения. Такой способ загрузки обеспечивает максимальную эффективность за счет отсутствия накладных расходов, связанных с упаковкой параметров вызова (marshalling), передачей упакованного блока данных в адресное пространство сервера с помощью некоторого транспортного механизма (сетевого протокола или буферов разделяемой памяти) и диспетчеризацией вызова обработчика запроса на серверной стороне.

Архитектура Yaffil Classic дает возможность приложениям использовать внутрипроцессный сервер Yaffil. Такое использование часто называют встраиваемым (embedded), подразумевая легковесность и упрощение тиражирования прикладных систем. По сравнению с традиционным сервером, внутрипроцессный Yaffil не требует запуска дополнительного процесса сервера или инсталляции служб NT. Приложению для работы требуется всего лишь одна библиотека динамической загрузки (DLL). Общий объем исполнимых модулей при этом также сокращается.

Однако встраиваемое использование подразумевает некоторые (возможно, значительные для вашего приложения) ограничения.

Встраиваемый сервер может использоваться только в однопоточных приложениях. Существующее ядро InterBase/Firebird/Yaffil не является безопасным для использования из нескольких потоков (thread-safe). Глобальные структуры данных сервера не защищены от одновременного изменения; кроме того, внутри ядра широко используется локальное состояние потока. Таким образом, поведе-

ние сервера будет непредсказуемым при вызове функций сервера с нескольких потоков одновременно, а также при использовании соединений, первоначально открытых в другом потоке.

Если вы разрабатываете программы, работающие в среде сервера приложений или Web-сервера, таких? как COM+ или IIS, то встраиваемый сервер для вас также непригоден, поскольку подобные среды используют собственное управление потоками.

Конфигурация безопасности для базы данных

При нахождении кода сервера в составе клиентского приложения необходим полный доступ к файлу базы данных. В то же время нельзя гарантировать разграничение доступа на основе разрешений SQL к объектам БД. Поскольку код приложения имеет физическую возможность обращаться к любой области базы данных, ограничения SQL являются всего лишь джентльменскими соглашениями.

С другой стороны? между приложением и базой данных нет посредников, таких? как сетевые устройства и средства межпроцессного обмена данными. Поэтому нет необходимости использовать средства защиты данных при клиент-серверном взаимодействии.

Использование сервера Yaffil внутри процесса

С точки зрения прикладной программы различие между встраиваемым сервером и обычным удаленным клиентом заключается в имени библиотеки динамической загрузки (DLL), связываемой с программой. Как известно, обычные приложения используют библиотеку GDS32.DLL, как правило, устанавливаемую в системный каталог Windows. Существование нескольких разных библиотек с одним именем может привести к путанице, особенно если подобная библиотека находится в пути доступа, общего для всех приложений. Версии 4.x InterBase CS, выпущенные фирмой Борланд, используют библиотеку сервера, которая также имеет имя GDS32.DLL.

Yaffil CS реализован в библиотеке YAENG32.DLL, имеющей интерфейс, идентичный GDS32.DLL. Поэтому использовать встраиваемый Yaffil CS можно в приложениях, написанных на IB API или Embedded SQL с помощью указания библиотеки импорта YAENG32.LIB на этапе связывания (linking).

Другая возможность использования нужной библиотеки состоит в динамической загрузке ее во время выполнения приложения.

Если же вы пользуетесь компонентами доступа или драйверами, не позволяющими указывать имя используемой библиотеки, вам остается только один выход – скопировать модуль YAENG32.DLL под именем GDS32.DLL. Не забудьте поместить этот файл в каталог, в котором находится исполнимый (.exe) модуль программы.

Эффективное взаимодействие процессов архитектуры Classic Server

В архитектуре Classic Server несколько серверных процессов совместно работают с одной базой данных, осуществляя координацию своих действий через разделяемую таблицу блокировок. Взаимодействие процессов на версиях

InterBase, работающих под операционными системами Unix, осуществляется с использованием механизма сигналов, при этом сигнал передается не напрямую, а через специальный процесс менеджера блокировок. Такой механизм быстро становится неэффективным при увеличении числа процессов, все большая часть времени процессора тратится на доставку и обработку сигналов.

В отличие от сигналов, в Yaffil на платформе Windows NT/2000 для обмена сообщениями о блокировках используются объекты синхронизации без обращений к ядру ОС. Во время активной работы сервера между процессами передается большое число таких сообщений, в результате заметно снижаются расходы на межпроцессное взаимодействие.

Изменения оптимизатора, направленные на совместимость

Благодаря улучшениям оптимизатора исключена ситуация, когда автоматически сгенерированный план запроса оказывается неверным, в то время как на более старых версиях он был правильным.

Yaffil Classic Server – замена InterBase Classic 4.0

InterBase CS 4.0 для операционной системы Windows NT до сих пор используется в системах, несущих большую нагрузку. Переход на более новые версии был невозможен в связи с тем, что архитектура Super Server недостаточно пригодна для работы с большим числом одновременных подключений, в то время как версии сервера архитектуры Classic Server перестали выпускаться Borland, начиная с версии 4.2.

Yaffil Classic Server – лучший вариант обновления сервера, сохраняет все преимущества архитектуры Classic, с лучшей производительностью и надежностью, обладает новыми возможностями последних версий линейки Interbase.

Миграция баз данных на Yaffil и обратно

При разработке сервера Yaffil большое внимание было уделено безболезненному переходу со всех версий линейки InterBase, начиная с версии 4.0. Это вызвано тем, что в настоящее время существует большое число инсталляций устаревших версий (4.2, 5.6) по причине затрудненного переноса приложений на новые версии InterBase. Проблемы состоят в недостаточной обратной совместимости более новых версий InterBase.

В то же время при использовании сервера Yaffil как обновления для InterBase версий 4.x, 5.x возможные проблемы сведены к минимуму. Как правило, приложение сразу успешно работает без переделок.

Перечислим изменения Yaffil, направленные на обратную совместимость:

- Поддержка баз данных, созданных в предыдущих версиях InterBase.
- Режим обратной совместимости.

Режим обратной совместимости

Данный режим включается установкой параметра конфигурационного файла LEGACY_DIALECT1 в 1. После этого для клиентов диалекта 1 компилятор SQL выражений будет поддерживать только возможности, существовавшие в InterBase

версии 5.x. Поведение сервера при работе с клиентами диалектов 2 и 3 не меняется. Режим совместимости приводит к "освобождению" следующих ключевых слов: COLUMN CURRENT_USER ROWS_AFFECTED CONNECTION_ID DESCRIPTOR SKIP CURRENT_DATE EXTRACT SUBSTRING CURRENT_ROLE FIRST TRANSACTION_ID CURRENT_TIME LEAVE TYPE CURRENT_TIMESTAMP RECREATE

Возможности, планируемые к реализации в следующих версиях

Интегрированная безопасность (NT Integrated Security)

Предполагается использовать принцип single sign-on, при котором при соединении пользователя к базе данных используется учетная запись домена NT, под которой он зарегистрирован в системе. Для передачи информации о пользователе используется протокол аутентификации выбранного модуля безопасности (Security Package), например NTLM в системах с доменами типа NT4 или Kerberos для доменов Windows 2000. Далее на уровне сервера учетная запись операционной системы отображается на имя пользователя.

Для использования этой возможности необходимо вхождение серверного и клиентского компьютера в домен NT. При использовании интегрированной безопасности пароль пользователя не передается в открытом виде по сети, возможно шифрование и подпись (confidentiality и privacy) сетевого трафика.

Подобные принципы организации безопасности используются в MS SQL, Sybase SQL Studio Anywhere и других СУБД.

Асинхронный сервер и отмена выполняющихся запросов

В сервере Yaffil предполагается реализовать асинхронную модель сервера, при этом для передачи запросов на отмену операции будет использоваться основное соединение.

Клиентский интерфейс отмены запросов предполагается совместимым с аналогичным расширением API в версии InterBase 6.5.

Одновременный запуск нескольких копий сервера (multi-instancing)

Существующие версии InterBase/Firebird не допускают одновременный запуск нескольких процессов сервера. Причина этого в том, что сервер использует глобальные именованные объекты и структуры данных. При запуске сервера происходит проверка на наличие уже запущенного экземпляра, однако иногда такая проверка может дать сбой, например в случае запуска в терминальной сессии Windows NT/2000.

При эксплуатации СУБД часто необходимо иметь несколько экземпляров сервера на одном компьютере для целей тестирования новой версии системы без влияния на основную инсталляцию или для разграничения задач администрирования.

Для решения такой задачи планируется разделить конфигурацию серверов и пространства имен глобальных объектов. Каждый экземпляр сервера будет использовать отдельный порт протокола TCP или имя Named Pipe.

Отметим, что сервер Yaffil использует отличные от InterBase/Firebird именованные глобальные объекты. Таким образом, появляется возможность одновременного запуска и Yaffil и InterBase/Firebird.

Хранение конфигурации в системном реестре

Для платформы Windows системный реестр – основное средство хранения конфигурационной информации. Текстовый файл IBCONFIG, используемый сейчас, имеет недостаток, связанный с невозможностью задания сложной конфигурации, имеющей иерархическую структуру. Кроме того, не так-то просто приложениям анализировать этот файл и вносить изменения в него.

Следующие версии Yaffil будут использовать системный реестр для хранения конфигурации.

Процедурный язык в SQL запросах

Традиционно InterBase использует разные диалекты SQL для написания текста запросов, принимаемых с клиента, и для описания текста хранимых процедур и триггеров. Некоторые типы операторов доступны только в одном из этих диалектов, в то время как другие конструкции имеют разный синтаксис в каждом случае.

В сервере Yaffil предполагается ввести возможность выполнения процедурных команд SQL в запросах, передаваемых с клиента, включая использование локальных переменных, операторы IF, WHILE, конструкции SELECT INTO и FOR SELECT.

Большие индексы

Планируется значительно увеличить максимальный размер ключа индекса с нынешних 128–256 (в зависимости от типов данных) байт в InterBase/Firebird. Как известно, при создании индексов по текстовым полям с национальным порядком сортировки (COLLATION) на каждый символ отводится 3 байта. Тем самым реальная максимальная длина индекса в настоящее время составляет 83 символа, что недостаточно для многих приложений.

Заключение

СУБД Yaffil 1.0 представляет собой бурно развивающуюся СУБД, основанную на исходных кодах InterBase 6.0 и Firebird 1.0.

Yaffil обладает лучшей производительностью по сравнению с InterBase 6.0 и Firebird 1.0 и предоставляет новую функциональность, сохраняя при этом совместимость с Borland InterBase 4–6.5 и Firebird 1.0.

Yaffil существует в варианте с классической архитектурой, что позволяет эффективно использовать его в многопользовательских приложениях, а также на SMP системах.

Несмотря на молодой возраст и большое число нововведений, Yaffil версии 1.0 в настоящее время является достаточно стабильным сервером и достойно представляет вклад российских разработчиков в создание семейства наследников InterBase 6.0 Open Edition.

InterBase 7

Семерка – первый шаг нового семейства

Чем InterBase 7 отличается от своих предшественников? Вот главный вопрос, на который отвечает эта глава.

Прежде всего надо отметить, что помимо непосредственно технических новшеств и улучшений в InterBase 7 был введен ряд изменений "политического" характера, направленных на разрушение совместимости с Firebird. Можно предположить, что это было сделано в связи с планами компании Borland выпускать новые версии своих продуктов, в том числе и InterBase, не реже чем раз-два в год. Таким образом, InterBase 7 лишь первый шаг в этом направлении, и можно с большой уверенностью сказать, что в 2003 году 8-я версия InterBase принесет нам много сюрпризов.

Но пока давайте сосредоточимся на технических подробностях 7-й версии InterBase.

Распараллеливание на несколько процессоров

Многие разработчики представляют себе понятие распараллеливания поразному, поэтому надо внести ясность, что имеется в виду под этим термином в случае InterBase 7. Прежде всего уточним, что речь идет о выполнении SQL-запросов, посылаемых пользователем. И тут обычно начинаются разночтения.

Одни считают, что распараллеливание – это когда разные части одного и того же SQL-запроса обрабатываются на различных процессорах. Другие считают, что распараллеливание – это когда разные соединения с пользователями "рассаживаются" по разным процессорам и все запросы от пользователя внутри них независимо выполняются на том процессоре, на котором выполняется обработка данного соединения (такой подход реализован в архитектуре Classic Server).

Однако в архитектуре SuperServer, реализованной в InterBase 7, в чистом виде не используется ни тот ни другой тип распараллеливания.

Чтобы понять, как InterBase 7 использует несколько процессоров, давайте немного углубимся в его архитектуру и рассмотрим процесс обслуживания запросов пользователя.

Когда пользователь посылает запрос на соединение, то этот запрос обрабатывается глобальным менеджером соединений и в случае правильности имени пользователя и пароля регистрируется в списке обслуживаемых клиентов.

Отдельного потока (thread) для индивидуального обслуживания только что подсоединенного клиента не запускается. Это легко можно проверить, написав простенькое приложение, которое откроет несколько сотен соединений с сервером, – и если эти соединения простаивают, то информация о процессе ibserver.exe в Task Manager показывает, что открыто всего 5–8 потоков.

Далее, как только подсоединенный пользователь пожелает выполнить какой-либо SQL-запрос, сервер запускает поток, который обслужит данный запрос. После обслуживания данного запроса поток через некоторое время завершает свою работу. Если этот же пользователь снова выполнит другой SQL-запрос, то

совершенно не обязательно, что его будет обслуживать тот же самый поток, что и в первый раз.

На самом деле, потоки не запускаются/уничтожаются исключительно по желанию пользователя – InterBase отслеживает среднюю загрузку/частоту выполнения SQL-запросов и держит постоянно открытым *пул потоков*, содержащий оптимальное число потоков, готовых обработать запросы пользователей.

Такая схема работы не только чрезвычайно экономична – позволяет держать оптимальное количество потоков, но и позволяет (точнее сказать – создает предпосылки для выполнения) выполнять в одном соединении несколько параллельных запросов (например, MSSQL этого не позволяют, требуя открытия отдельного соединения для параллельного выполнения SQL-запросов с одного клиента).

Теперь читатели наверняка догадались, как работает распараллеливание в InterBase, – каждый запускающийся поток будет привязан к наименее загруженному процессору, который и обработает (целиком!) SQL-запрос пользователя. Таким образом, разные запросы от одного пользователя могут быть выполнены (в том числе и параллельно) на разных процессорах.

Итак, 7-я версия InterBase – это масштабируемый сервер архитектуры SuperServer. Если вы внимательно читали главу "Classic и SuperServer", то знаете, что главным недостатком архитектуры SuperServer была невозможность эффективно использовать несколько процессоров. В 7-й версии InterBase эта проблема была разрешена, и теперь у InterBase есть возможность работать на многопроцессорных серверах.

Не имея исходных кодов семерки, можно лишь с относительно высокой долей вероятности предположить, что разработчики этой версии пошли по пути оптимизации существующего кода, т. е. InterBase 7 не переписывался "с нуля". Учитывая огромный объем исходного кода сервера, очевидно, что изменить InterBase 6.x для поддержки распараллеливания, а затем отследить и протестировать эти изменения – это грандиозная работа, за которую пока не взялся никто, кроме разработчиков из Borland.

Как бы то ни было, InterBase 7 может использовать мощь нескольких процессоров одновременно, а также более эффективно обрабатывает одновременно выполняющиеся SQL-запросы даже на однопроцессорных машинах. Это значительно расширяет сферу его применения и увеличивает ту долю рынка, которую InterBase может забрать у других, гораздо более "монстровидных" СУБД.

InterBase 7 предоставляет средства распределения загрузки по нескольким процессорам. Например, вы можете выделить InterBase только 2 из 4 процессоров, а оставшиеся загрузить другими, не менее важными делами. Для управления привязкой InterBase к процессорам используется специальный параметр CPU_AFFINITY в файле конфигурации сервера ibconfig. В документации к InterBase 7 приведена таблица, показывающая, какое значение должен иметь данный параметр для привязки к желаемым процессорам, и мы здесь ее повторять не будем.

Также в ibconfig появился параметр MAX_THREADS, который устанавливает число одновременно открытых в пуле потоков, готовых к обслуживанию запросов. Изменение этого параметра позволяет управлять загрузкой процессоров –

чем меньше потоков, тем меньше процессорного времени будет потреблять InterBase. Для однопользовательских приложений рекомендуется установить этот параметр в 1. Максимальное значение этого параметра – 100.

Важно отметить, что MAX_THREADS не ограничивает число возможных соединений (т. е. обслуживаемых клиентов) с сервером, а лишь устанавливает максимальное значение активных потоков. Таким образом, можно "подсказать" серверу, чтобы он не слишком "экономил" и не закрывал потоки при отсутствии загрузки, т. е. другими словами, был готов к резкому увеличению количества клиентов.

Мониторинг состояния сервера

Любители исследовать исходный код InterBase 6 часто говорят, что в нем скрыта масса интереснейших идей, которые были не доведены до конца по каким-либо причинам. Одной из таких идей, извлеченных и реализованных в InterBase 7, является мониторинг внутреннего состояния сервера.

В InterBase предыдущих версий сервер производил постоянный мониторинг своего состояния – отслеживал выполняющиеся запросы, подключенных пользователей и т. д. Но все эти данные были для внутреннего использования – пользователи не могли получить к ним доступ. Теперь ситуация изменилась – в InterBase 7 появился удобный интерфейс для доступа к данным о мгновенном состоянии сервера.

InterBase 7 предоставляет механизм "временных системных таблиц" для доступа к данным о своем состоянии. Не надо путать эти временные таблицы с временными таблицами в других СУБД, которые используются для оптимизации выполнения SQL-запросов.

В данном случае временные системные таблицы лишь представление мгновенного снимка состояния сервера и баз данных. Чтобы избавить разработчиков приложений баз данных от необходимости использовать специальные вызовы API для получения (и даже изменения!) информации о состоянии сервера, в семерке создан SQL-интерфейс для этих целей. Не правда ли, очень удобно – формируете стандартными средствами SQL-запрос и получаете нужную статистику в виде привычного набора данных.

Выполняя запросы к временным таблицам, можно получить свежие (на момент выполнения запроса) данные о том, какие запросы выполняет конкретный подключенный пользователь, какие таблицы использует. Чтобы обновить полученные данные, надо подтвердить транзакцию, в рамках которой выполнялся запрос к временным системным таблицам, и выполнить запрос снова.

Можно также узнать, какие запросы в данный момент выполняются на сервере и какой из этих запросов самый длительный. Помимо этого, существует еще множество видов информации, которую можно извлечь из системных таблиц.

Вот список временных системных таблиц для мониторинга в InterBase 7 с краткими описаниями, взятый с сайта www.ibase.ru:

Таблица 2. Системные таблицы для мониторинга в InterBase 7

Доступные системные временные таблицы	
Название таблицы	Что содержит
TMP$ATTACHMENTS	По записи на каждое соединение к базе данных

TMP$DATABASE	По записи на каждую базу данных, к которой вы подсоединились
TMP$POOL_BLOCKS	По записи на каждый блок памяти в каждом пуле
TMP$POOLS	По записи на каждый пул памяти
TMP$PROCEDURES	По записи на каждую выполненную в данном соединении процедуру
TMP$RELATIONS	По записи на каждую таблицу, к которой было обращение в данном соединении
TMP$STATEMENTS	По записи на каждый выполняемый в данный момент запрос, для всех соединений
TMP$TRANSACTIONS	По записи на каждую активную (или в состоянии limbo) транзакцию

Модификация системных таблиц

Помимо чтения статистики, есть ряд случаев, когда системные таблицы можно менять! Прежде всего, это тот важный случай, когда необходимо снять очень длительный, зависший запрос.

Чтобы влиять на состояние запросов, транзакций, соединений и т.д., нужно изменить столбец TMP$STATE в соответствующей временной системной таблице. Например, вы можете произвести следующие изменения:

- Отключить соединение.

```
UPDATE TMP$ATTACHMENTS SET TMP$STATE = 'SHUTDOWN'
WHERE (TMP$ATTACHMENT_ID = 12345)
```

- Принудительно отменить активную в данный момент или "застрявшую" 2PC (т. е. в состоянии in limbo) транзакцию.

```
UPDATE TMP$TRANSACTION SET TMP$STATE = 'ROLLBACK'
WHERE (TMP$TRANSACTION_ID = 12345)
```

- Остановить выполняемый в данный момент запрос.

```
UPDATE TMP$STATEMENTS
SET TMP$STATE = 'CANCEL'
WHERE (TMP$STATEMENT_ID = 12345)
```

Примеры получения статистики

Примерами использования системных таблиц для получения полезной статистики могут являться следующие запросы:

- 10 самых длительно выполняющихся запросов.

```
SELECT a.tmp$user, s.tmp$timestamp, s.tmp$sql, s.tmp$quantum
FROM tmp$statements s, tmp$attachments a
WHERE a.tmp$attachment_id = s.tmp$attachment_id
ORDER BY s.tmp$quantum DESC
ROWS 10;
```

- Активность пользователя SYSDBA.

```
SELECT TMP$USER, TMP$USER_IP_ADDR,
TMP$TIMESTAMP, TMP$STATE,
TMP$TRANSACTIONS, TMP$RECORD_SELECTS,
TMP$RECORD_INSERTS,
TMP$RECORD_UPDATES, TMP$RECORD_DELETES
FROM TMP$ATTACHMENTS
WHERE TMP$USER = 'SYSDBA'
```

Безопасность временных таблиц

Чтобы предотвратить возможность использования мощных возможностей временных таблиц в неблаговидных целях, на них введены ограничения прав доступа – по умолчанию таблицы видимы и могут изменяться только владельцем базы данных или SYSDBA. В случае необходимости можно выдать права на чтение (и только на чтение) для других пользователей – обычным образом, с помощью оператора GRANT и отобрать их с помощью REVOKE.

JDBC Type 4 DRIVER

Java-разработчики могут быть довольны – наконец у InterBase появился собственный "тонкий" JDBC-драйвер – InterClient 3.0. Это означает, что он не требует никаких дополнительных промежуточных программ вроде InterServer – достаточно просто подключить interclient.jar в свою строку CLASSPATH и работать с InterBase 7. Это значительно упрощает распространение Java-приложений баз данных InterBase.

Помимо упрощения распространения, InterClient 3.0 полноценно поддерживает пул соединений (Connection Pooling) и механизм источников данных (DataSource). Также InterClient 3.0 теперь поддерживает полноценную работу с Blob-полями InterBase 7.

Хочется отметить также превосходную интеграцию InterBase 7 с такими продуктами Borland, как JBuilder и Borland Enterprise Server. Эта связка СУБД, мощного средства разработки и application-сервера позволяет легко разрабатывать J2EE-приложения.

Некоторым недостатком InterClient 3.0 JDBC Type 4 является то, что он работает только с 7-й версией InterBase, и потому в приложениях, использующих более старые версии, работать не будет. Этот факт тем огорчительнее, что конкурент InterClient – Open Source JayBird от команды Firebird developers все еще находится в стадии бета-тестирования, и поэтому множество Java-разработчиков вынуждено будет пользоваться более старыми версиями InterClient – JDBC Type 3, которые используют промежуточную программу InterServer для работы с базой данных. Для таких разработчиков в поставку InterBase 7 включен хорошо знакомый InterClient JDBC Type 3.

Новая структура данных на диске: ODS11

Для поддержки нововведений базы данных, созданные (или восстановленные) в InterBase 7, имеют новую версию внутренней структуры базы данных – On-Disk Structure (ODS). Новая версия ODS несовместима с прежними ODS. Это

значит, что старые версии InterBase и клоны InterBase Open Source (Firebird и Yaffil) не будут работать с базами данных, имеющими ODS11.

Миграция баз данных на новую ODS возможна только через backup/restore – по тому же самому принципу, что описан в главе "Миграция". Следует также отметить, что в InterBase 7 все же поддерживаются базы данных 1-го диалекта, хотя при выпуске 6-й версии объявлялось, что далее диалект 1 поддерживаться не будет. Однако очевидно, что все еще очень много пользователей используют базы данных в 1-м диалекте и не могут по различным причинам легко перейти на 3-й диалект. Поэтому InterBase 7 поддерживает как 3-й, так и 1-й диалект.

Ниже мы коротко рассмотрим остальные нововведения, напрямую ответственные за появление 11-й версии On-Disk Structure.

Новый тип данных: BOOLEAN

InterBase теперь поддерживает тип данных BOOLEAN в соответствии со стандартом SQL99. Поля и переменные типа BOOLEAN могут принимать значения TRUE/FALSE/UNKNOWN (да-да, и здесь используется трехзначная логика, как и везде, где есть понятие неопределенного значения). Размер BOOLEAN – 32 бита.

Чтобы создать в таблице поле типа BOOLEAN, достаточно написать что-то вроде этого:

```
CREATE TABLE Table1(MyBOOL BOOLEAN)
```

Возможным значениям типа BOOLEAN – TRUE, FALSE и UNKNOWN соответствуют целые значения 0, 1 и неопределенное значение NULL.

Новые ключевые слова

В Interbase 7 появились новые ключевые слова, связанные с вышеупомянутым типом BOOLEAN:

```
BOOLEAN, TRUE, FALSE, UNKNOWN
```

Хочется отметить, что в предыдущей версии (6.5) были добавлены следующие ключевые слова:

```
ROWS, TIES, PERCENT
```

Их значение разъяснено в документации к InterBase 6.5.

Имена объектов длиной 68 символов

Возможная длина имен объектов в InterBase 7 теперь равна 67 символов вместо 31 символа ранее. Да, именно 67 – хотя в заголовке этого раздела написано 68, фактически хранится лишь 67 символов, а последний символ представляет собой завершающий 0.

Очевидно, чтобы воспользоваться данной возможностью, необходимо обновить версию клиентской библиотеки gds32.dll (а Java-разработчики должны использовать Type 4 драйвер).

Чтобы поддержать данное изменение длины имен объектов, была изменена структура XSQLDA. Надо сказать, что данное изменение весьма неприятно сказалось на клиентских библиотеках доступа к InterBase, таких, как IBX, FIBPlus,

dbExpress и т. д. Теперь, чтобы перевести ваше программное обеспечение под InterBase 7, понадобится перекомпилировать существующие клиентские приложения с новыми версиями клиентских библиотек.

Новые функции API для работы с Blob и массивами

Были добавлены 10 новых функций InterBase API для поддержки длинных имен объектов. Ниже представлены новые вызовы API:

```
isc_array_gen_sdl2()
isc_array_get_slice2()
isc_array_lookup_bounds2()
isc_array_lookup_desc2()
isc_array_set_desc2()
isc_array_put_slice2()
isc_blob_default_desc2()
isc_blob_gen_bpb2()
isc_blob_lookup_desc2()
isc_blob_set_desc2()
```

Другие изменения в 7-й версии InterBase

SET TERM больше не нужен в isql

Как вы знаете из главы "Хранимые процедуры", для создания хранимых процедур и триггеров с помощью SQL-скриптов и интерпретатора isql необходимо предварять и завершать команды создания и изменения процедур и триггеров специальной командой смены разделителя.

В 7-й версии ликвидирована нужда в команде SET TERM – теперь интерпретатор SQL корректно обрабатывает указанные выше команды и не выдает ошибку, которая ранее являлась просто ночным кошмаром для начинающих (во всяком случае, для тех из них, что не любят читать документацию).

Определение версии клиента

Некоторые клиентские библиотеки и драйверы могут иметь необходимость определять версию клиентской библиотеки Interbase. Для этого введены три новые функции API:

```
isc_get_client_version(),
isc_get_client_major_version(),
isc_get_client_minor_version().
```

Безопасность внешних таблиц. Параметр EXTERNAL_FILE_DIRECTORY

В определенных условиях внешние таблицы (external table) могут быть источником проблем в безопасности. Известно, что в предыдущих версиях

InterBase, используя механизм внешних таблиц, можно было выкрасть всю базу целиком, от первого до последнего байта!

Здесь мы, конечно, не будем приводить этот способ, но скажем, что в InterBase 7 наконец разрешили эту серьезную проблему путем введения ограничений на возможное расположение системных таблиц. Теперь внешние таблицы должны удовлетворять следующим условиям:

- Внешние таблицы должны находиться в каталоге <interbase_home>/ext. InterBase будет сначала искать внешние таблицы в этом каталоге.
- Если внешняя таблица находится не в каталоге /ext, то путь к каталогу, где она находится, нужно указать в ibconfig при помощи параметра EXTERNAL_FILE_DIRECTORY. Параметр можно указывать несколько раз для всех каталогов, где могут находиться внешние таблицы.
- Введение этих ограничений позволяет значительно ограничить доступ злоумышленника к данным внутри базы данных.

Единое имя файла параметров InterBase

Теперь файл параметров InterBase 7 имеет единое имя как для платформы Windows, так и для Linux – ibconfig.

Рекомендуемое расширение для файлов баз данных – *.ib

Теперь рекомендуемым разрешением для файлов баз данных становится ib – вместо привычного gdb. Прежде всего эта смена расширения связана с тем, что при использовании InterBase на Windows XP эта операционная система распознает файлы с расширением gdb как нечто системное и пытается при любых его изменениях сделать резервную копию. В результате чего производительность колоссально снижается. Конечно, можно эту функцию Windows XP отключить с помощью определенных манипуляций, однако для предупреждения проблем с тиражируемыми приложениями, которые должны ставиться автономно, было принято решение изменить расширение.

Конечно, InterBase 7 по-прежнему будет работать с базами данных, имена файлов которых имеют расширения любых видов и вообще не имеют расширения, но рекомендуется все же использовать *.ib.

Новое имя базы данных пользователей

Отныне база данных пользователей isc4.gdb отходит в прошлое, и вместо нее по умолчанию будет использоваться база данных *admin.ib*. По сути, она выполняет идентичные функции.

Правда, наименование базы данных пользователей теперь не является жестко заданным – используя параметр ADMIN_DB, вы можете задать любое приятное вашему слуху имя базы данных пользователей InterBase.

Заключение

Выход InterBase 7 показал всем, что у компании Borland есть еще "порох в пороховницах", и отказываться от этой СУБД она не намерена.

Нельзя сказать, что InterBase 7 поразил воображение пользователей своими нововведениями, но он сделал очень хороший задел на будущее – прежде всего, благодаря эффективной поддержке многих процессоров.

В любом случае InterBase 7 стал родоначальником новой ветки СУБД на основе InterBase 6.0 и первой значительно отличающейся от семейства InterBase 6 Open Source/Firebird/Yaffil окончательной версией (release) сервера. Часть его нововведений можно расценивать именно как политическую, направленную на четкое позиционирование и отгораживание от других клонов. Это положительная тенденция, потому что вместо группы сходных серверов с неясными различиями мы, пользователи СУБД, сможем получить несколько отличных серверов баз данных!

Firebird 1.5 – Open Source DBMS

Материал для данной главы предоставлен Дмитрием Владимировичем Емановым

Прошел год с момента выпуска первой версии СУБД Firebird (12 апреля 2002 года) и, таким образом, с момента официального вхождения продукта в мир Open Source и начала завоевания собственного авторитета. Но это уже в прошлом, нас же больше интересует состояние дел на текущий момент, а также планы развития проекта на ближайшее будущее. Сейчас с полной определенностью можно сказать, что этот год прошел не зря для Firebird. Была проделана большая работа, результаты которой можно наблюдать сегодня. Ниже будет подробно рассказано о том, что уже сделано и что еще предстоит сделать участникам проекта для достижения поставленных ими целей, главной из которых является выпуск в свет версии 1.5 в начале 2003 года.

Продолжение линии 1.0

Сразу после выхода первого релиза был начат процесс анализа реакции пользователей и исправление найденных проблем. Как результат, вскоре были выпущены несколько пострелизных сборок, одна из которых (821) стала в итоге версией 1.0.1, а сборка 908 – 1.0.2.

В версию 1.0.2 вошли несколько исправлений ошибок, основными из которых являются:

- корректная реализация 64bit I/O (поддержка больших файлов) для UNIX-систем;
- решение проблемы неправильной строки соединения (опять же для UNIX-систем);
- возможность работы с типом INT64 в массивах;
- устранение проблем с механизмом двухфазной фиксации транзакций.

Эти изменения позволят пользователям линии 1.0 решить их насущные проблемы и чувствовать себя уверенно вплоть до выхода новой версии. Таким образом, релиз 1.0.2 ставит точку на развитии линии 1.0 и позволяет окончательно сконцентрировать внимание на разработке следующих версий.

Версия 1.5 – эволюция или революция?

Так что же собой представляет новая версия Firebird? Чтобы получить ответ на этот вопрос, позволим себе небольшой экскурс в историю. Летом 2001 года администраторами и ведущими разработчиками проекта было принято решение о серьезной переработке исходного кода сервера, доставшегося в наследство от корпорации Borland (в то время Inprise). Это было вызвано следующими соображениями:

- необходимость инкапсуляции ключевых объектов ядра СУБД для более качественной их защиты и независимых блокировок (что жизненно важно для реализации в сервере полноценной поддержки SMP);
- обеспечение удобства программного расширения подсистем сервера;
- реализация более гибкого и контролируемого управления памятью;

- обеспечение полного контроля над исключительными ситуациями внутри сервера;
- возможность использования в разработке современных библиотек и шаблонов программирования.

Одним из методов достижения этих целей было выбрано использование C++ в качестве языка программирования (в оригинале сервер написан на C) и соответствующее портирование существующего исходного кода. Так как этот процесс является достаточно трудоемким, а ветка версии 1.0 в тот момент времени развивалась очень активно, решили создать отдельную ветку для разработки новой версии сервера (2.0), что обеспечило участникам проекта бесконфликтную параллельную работу над двумя версиями – текущей и будущей. В соответствии с планом, ветку назвали firebird2 – как олицетворение независимости от истоков и обещание заметных нововведений. К моменту выхода версии 1.0 портирование кода было завершено, также уже была реализована новая обработка исключительных ситуаций и перепроектирован менеджер памяти. Вскоре после этого был разработан новый универсальный механизм подключения к серверу внешних модулей (plugins), который может быть использован для поддержки неограниченного числа определяемых пользователем языковых кодировок, реализации механизмов безопасности посредством PAM (pluggable authorization modules), сжатия или шифрования сетевого трафика и т. п.

Так как вся эта работа отняла достаточно много времени, было принято решение о стабилизации кода ветки firebird2, внесении в нее нескольких функциональных улучшений (преимущественно в языке SQL), об исправлении ряда ошибок и выпуске данного кода как версии 1.5.

Таким образом, версия Firebird 1.5 является промежуточной между классической 1.0 и революционной 2.0 и представляет собой первый шаг в направлении разработки радикально улучшенного сервера.

Достигнутые результаты

Большая часть запланированных функциональных возможностей была реализована к сентябрю 2002 года, и в конце сентября были выпущены первые неофициальные сборки новой версии (snapshots) для платформы Win32. Приблизительно через месяц была выпущена первая альфа-версия, которая, будучи нестабильной и предназначенной исключительно для технологического тестирования, тем не менее давала возможность ознакомиться с особенностями новой версии и предоставить производителям стороннего ПО условия для адаптации под нее своих продуктов. Помимо платформы Win32, были выпущены сборки новой версии сервера для Linux, MacOS X (Darwin) и SINIX-Z.

На момент выхода данного издания книги в свет код новой версии должен быть заблокирован для инноваций и должны быть выпущены так называемые релиз-кандидаты, т. е. финальные сборки, дающие возможность оценить готовность версии 1.5 к официальному опубликованию.

Отличительные особенности новой версии

Итак, что же нового приготовили нам разработчики Firebird? Ниже приводится список нововведений с их кратким описанием, сгруппированный по функциональным категориям. Следует заметить, что не все из приводящегося списка в настоящий момент полностью реализовано, поэтому он может несколько отличаться от официального документа Release Notes, включенного в дистрибутив новой версии.

Дистрибутив

Первое, на что стоит обратить внимание, это отличие в именах файлов. Исполняемый файл сервера теперь называется либо fbserver (в случае архитектуры SS), либо fb_inet_server (в случае CS). Следует отметить, что теперь сервер с классической архитектурой доступен и для пользователей платформы Win32. Клиентская библиотека носит имя fbclient. В дистрибутивы для Win32 входит также библиотека gds32.dll, которая теперь является просто "заглушкой", перенаправляющей все вызовы к fbclient.dll, и поставляется исключительно для совместимости с существующими приложениями. Все новые приложения, ориентирующиеся на использование этой и последующих версий Firebird, должны обращаться к новой клиентской библиотеке (т. е. fbclient). Стандартная библиотека языковых кодировок теперь называется fbintl. Также изменились имена некоторых других файлов.

Вариант сервера с классической архитектурой для платформы Win32 несколько отличается от аналогичных дистрибутивов для UNIX-систем. Главная особенность – это единый пакет, включающий в себя SS и CS сборки сервера, причем оба варианта скомпонованы статически. Это позволило избежать реализации кода сервера в библиотеке, разделяемой всеми процессами, и обеспечить четкое разделение клиентской и серверной подсистем. Результатом этого стала независимость стандартных инструментов командной строки (isql, gbak, gfix и т. д.) от дистрибутива сервера, а также возможность смены архитектуры установленного сервера "на лету", т. е. без переустановки дистрибутива.

Далее, версия сервера для платформы Win32 теперь использует другой ключ реестра (HKLM\Software\FirebirdSQL\Firebird). В случае отсутствия данного ключа сервер все равно будет работоспособен, его базовая директория при этом будет определяться физическим расположением файлов сервера. Следовательно, в простейшем случае (при запуске сервера как приложения, т. е. с ключом командной строки "-a") установка вообще не требуется – достаточно просто скопировать файлы сервера в отдельный каталог и запустить его.

Теперь можно иметь возможность запуска сервера Firebird параллельно с сервером IB/FB1 на одном компьютере, а также запуска нескольких копий сервера (все это при условии работы на разных портах, что настраивается перед запуском сервера).

Помимо собственно СУБД, проект Firebird включает также драйверы ODBC, JCA-JDBC (Type 4) и .NET-провайдер, которые распространяются вместе с сервером или отдельно.

Реализация языка SQL

Здесь можно назвать сразу ряд новых возможностей, как-то: новые команды DDL (RECREATE <object>, CREATE OR ALTER <object>, ALTER VIEW), полноценную поддержку больших целых чисел (встроенный тип BIGINT, реализованный как 64-битное целое), универсальные триггеры на набор операций, возможность использования выражений в параметрах процедур, группировка по встроенным функциям (например, EXTRACT) и т. д.

Добавлены новые встроенные функции – CASE, COALESCE и NULLIF (все они являются частью SQL-стандарта). CASE является базовой для этого ряда функций и реализует условную выборку значений из списка, например:

```
SELECT CASE WHEN (O.TYPE = 0) THEN 'Доход' ELSE 'Расход' END
FROM OPERATIONS O
```

или так:

```
SELECT CASE O.TYPE WHEN 0 THEN 'Доход' ELSE 'Расход' END
FROM OPERATIONS O
```

Стоит отметить, что у функции CASE может быть любое количество аргументов, т. е. она не ограничивается только простейшей проверкой типа "ЕСЛИ-ИНАЧЕ", например:

```
SELECT CASE O.TYPE WHEN 0 THEN 'Доход' WHEN 1 THEN 'Расход'
ELSE '---' END
FROM OPERATIONS O
```

Функция COALESCE является упрощением CASE для проверки на NULL, например:

```
SELECT COALESCE(O. STATUS, '---')
FROM OPERATIONS O
```

Результатом COALESCE будет являться первый аргумент, если он не NULL, или второй аргумент в противном случае. Функция NULLIF также является упрощением CASE, но для немного другого случая:

```
SELECT NULLIF(O.STATUS1, O.STATUS2)
FROM OPERATIONS O
```

Результатом NULLIF будет являться NULL, если оба аргумента равны, или первый аргумент в противном случае.

Кроме встроенных функций, добавлены также новые системные переменные:

- CONNECTION_ID – идентификатор текущего соединения;
- TRANSACTION_ID – идентификатор текущей транзакции;
- ROWS_AFFECTED – количество записей, затронутых (т. е. добавленных, исправленных или удаленных) последним выполненным оператором;
- GDSCODE и SQLCODE – соответствующие коды исключений, перехваченных в блоках WHEN.

Переменные CONNECTION_ID и TRANSACTION_ID доступны во всех вариантах SQL, в то время как ROWS_AFFECTED, GDSCODE и SQLCODE – только в PSQL (т. е. в хранимых процедурах и триггерах). Клиентское приложение может

получить значения упомянутых системных идентификаторов (CONNECTION_ID и TRANSACTION_ID) через запрос, аналогичный следующему:

```
SELECT TRANSACTION_ID FROM RDB$DATABASE;
```

Также расширены возможности работы с исключениями в хранимых процедурах и триггерах с помощью команды EXCEPTION:

- EXCEPTION (без параметров) – заново инициирует перехваченное в WHEN-блоке исключение;
- EXCEPTION <name> <msg> – инициирует исключение с текстом, определяемым выражением msg.

Еще одним важным нововведением является поддержка точек сохранения (savepoints), действующих внутри транзакции. Этот стандартный механизм позволяет откатить не всю транзакцию, а только часть ее, выполненную после указанной точки сохранения. Для этого используются следующие SQL-команды:

```
SAVEPOINT <name>
```

и

```
ROLLBACK [WORK] TO [SAVEPOINT] <name>.
```

В настоящий момент эти команды недоступны в PSQL и могут быть вызваны только с клиентской стороны, при этом они выполняются в контексте клиентской транзакции.

Помимо всего перечисленного, интерес может представлять и возможность динамического выполнения SQL-выражений в хранимых процедурах и триггерах. Для этого используется новая команда EXECUTE STATEMENT <stmt>, где единственным параметром является строка, содержащая выполняемое SQL-выражение. Например:

```
stmt_var = 'update my_table set flag = 0 where parent_id = ' ||
cast (:param as varchar(10));
EXECUTE STATEMENT stmt_var;
```

Этим новые возможности в SQL не ограничиваются, но их описание выходит за рамки данной главы.

Расширение механизма событий

Механизм оповещения о событиях (event alerters) является способом асинхронной передачи информации с сервера клиенту. Для этого клиент регистрирует свой интерес в конкретных именованных событиях и переходит в режим ожидания оповещений от сервера, который посылает ему их с помощью команды POST_EVENT, доступной в хранимых процедурах и триггерах. События посылаются сервером не сразу, а в момент подтверждения транзакции. При этом каждое событие имеет внутренний счетчик инициации, т. е. если в течение транзакции команда POST_EVENT для данного события была вызвана несколько раз, то клиенту будет доставлено оповещение только однажды, но его счетчик будет содержать количество инициаций. Для работы с событиями на клиентской стороне используются следующие функции API: isc_event_block, isc_que_events, isc_wait_for_ event и isc_event_counts.

Но максимально полному использованию этого механизма мешала фиксация имени события, т. е. клиент всегда должен ожидать событие с точно таким именем, которое инициирует сервер, а также невозможность передать параметры вместе с событием. В новой версии Firebird механизм событий был расширен возможностью обработки так называемых параметризованных событий, когда каждое инициируемое событие может иметь аргумент, который будет передан на клиентскую сторону вместе с событием. Тогда сервер может инициировать событие следующим образом (например, в триггере):

```
POST_EVENT 'MY_EVENT', NEW.OPER_TYPE || '_ID=' || CAST(NEW.ID
AS VARCHAR(10));
```

Для получения списка аргументов событий с их собственными счетчиками инициации была введена новая функция API: isc_event_params.

Такая возможность позволяет максимально гибко реализовать "интеллектуальное" обновление данных в клиентском приложении, внешнюю репликацию и ряд других задач.

Производительность

В разработке новой версии были предприняты некоторые шаги по повышению быстродействия сервера. Во-первых, небольшой прирост производительности дала переработка некоторых подсистем сервера, в частности оптимизация менеджера памяти.

Был улучшен алгоритм работы оптимизатора запросов, в частности в следующих направлениях:

- использование индексов в подзапросах с агрегатами;
- игнорирование "плохих" индексов, т. е. индексов, использование которых может привести лишь к замедлению выполнения запроса;
- более качественный анализ селективности индексов при построении плана;
- оптимальное построение плана в случае доступности одно- и многосегментных индексов;
- более совершенное использование индексов в условиях типа "OR", особенно в случае сложных предикатов (в частности, сложных неявных соединений).

Кроме этого, были внесены изменения в код сервера, приводящие к эффективному использованию оперативной памяти в процессе сортировки (режим использования памяти настраивается через файл конфигурации). Реализована также новая стратегия работы сортировщика, оптимизированная для некоторых видов запросов, требующих неполной выборки данных.

Известно, что текущая реализация архитектуры SS имеет ряд серьезных недостатков, одним из которых является блокирование сервера долго выполняющимися запросами, что приводит к практической невозможности работы в этот момент других пользователей. Эта проблема была частично решена в версии 1.5 путем доработки внутреннего планировщика потоков, базирующейся на более гибком управлении интервалами работы потоков и их приоритетами. В результате заметно улучшена реакция сервера при одновременной работе нескольких пользователей с высокой нагрузкой.

Было добавлено несколько новых системных индексов (т. е. индексов на системные таблицы), что позволило более быстро выполнять запросы к метаданным в случае сложной схемы и, следовательно, более оперативно заполнять кэш метаданных при первом обращении к ним.

Все вышесказанное привело к заметному увеличению производительности. Например, выполнение серии промышленных тестов TPC-R (уже упоминавшейся в этой книге) показало почти трехкратное превосходство новой версии по сравнению с линией 1.0. Разумеется, это не значит, что выполнение абсолютно всех запросов будет осуществляться быстрее, но позволяет рассчитывать на улучшение производительности в ряде случаев.

Конфигурирование

Эта версия сервера содержит новый менеджер конфигурации, единый для всех платформ. Все настройки теперь хранятся в файле firebird.conf. Этот файл содержит значительно больше параметров, чем ibconfig в предыдущих версиях. Например, в него вынесены параметры управления памятью, использованием и настройкой сетевых протоколов, режимами совместимости по новым возможностям, настройками безопасности и т. п.

Кроме того, в версии 1.5 реализован механизм "псевдонимов" (aliases) баз данных, т. е. возможность подключения к БД без знания физического пути к соответствующему файлу. Список псевдонимов хранится в файле aliases.conf, расположенном в корневом каталоге установки сервера. Запись о псевдониме выглядит следующим образом:

```
<aliasname> = <database pathname>
```

При этом строка соединения с сервером будет выглядеть следующим образом (в случае TCP/IP): <hostname>:<aliasname>. Это позволяет избавить клиентов от необходимости знать структуру файловой системы сервера и обезопасить сервер от несанкционированного доступа извне.

Firebird 2.0 – взгляд в будущее

Итак, выше мы узнали о том, что сделано в рамках версии Firebird 1.5, теперь самое время поговорить о том, что нам стоит ожидать от следующей версии, которая будет носить номер 2.0.

Новая версия ODS

Значительная часть запланированных изменений потребует реализации новой версии ODS (On-Disk Structure), что было признано неприемлемым для версии 1.5 (которая является полностью совместимой с IB6/FB1), поэтому введение новой ODS отложено до версии 2.0. Элементами новой ODS будут являться:

- увеличение размера полей, хранящих имена метаданных, что позволит обойти текущее ограничение в 31 символ;
- увеличение размера ряда внутренних идентификаторов до 64 бит, что позволит, например, хранить больше записей в одной таблице и обрабатывать большее количество транзакций без необходимости цикла backup/restore;

- более полная реализация безопасности метаданных на уровне SQL, что снимет сразу ряд имеющихся в настоящее время проблем в этой области;
- перенос информации о пользователях и группах из системной базы данных isc4.gdb внутрь оперативных баз данных для более качественной защиты тиражируемых продуктов;
- реализация признака корректности скомпилированного объекта базы данных и соответствующего механизма проверки (validation), что позволит более гибко контролировать зависимость объектов друг от друга и их работоспособность при обнаруженных нарушениях в цепочке зависимостей;
- возможность работы со значительно большими размерами ключа индекса;
- несколько новых типов данных и объектов базы данных.

Поддержка SMP

Несмотря на то что классическая архитектура сервера полностью использует возможности SMP-систем, она имеет ряд недостатков, которые накладывают слишком высокие требования на ресурсы сервера в случае очень большого количества соединений с базой данных. Как известно, корпорация Borland отказалась от развития архитектуры Classic и сосредоточила свои усилия на доработке кода SuperServer. В 2002 году мы увидели первый результат их работы – версию InterBase 7.0, в которой декларирована поддержка SMP.

Одной из целей разработки Firebird 2.0 также является полноценная поддержка SMP во всех вариантах сервера. Здесь можно пойти двумя путями: доработка кода SS (аналогично тому, что сделали в Borland) или объединение кода SS и CS с получением комбинированной архитектуры, позволяющей в зависимости от конфигурации работать либо в режиме разделяемого сервера с общим пулом потоков, либо в режиме выделенного сервера с одним процессом на каждое соединение, либо в одном из смешанных режимов (один сервер на N соединений, один сервер на каждую БД и т. п.). Оба варианта достаточно трудны в реализации, и разработчики проекта еще не приняли окончательного решения на этот счет, но пока наблюдается тенденция к выбору второго варианта (комбинированная архитектура). Это также позволит унифицировать исходный код и упростить внесение в него низкоуровневых изменений.

Средства мониторинга

Планируется введение ряда так называемых виртуальных таблиц, которые будут предоставлять пользователю или администратору доступ к внутренней информации сервера (подобный подход использован в InterBase 7.0).

Код, реализующий данные функциональные возможности, уже разработан участниками проекта, но необходимость тщательного проектирования виртуальных таблиц и тестирования существующей реализации не позволила включить его в версию 1.5, поэтому эти возможности станут доступными только в версии 2.0.

Дальнейшее развитие языка SQL

Развитие будет осуществляться по следующим направлениям:

- добавление новых или расширение возможностей существующих команд DDL, в частности по работе с доменами, признаками полей таблиц, кодом процедур и триггеров;
- реализация уникальных ограничений (unique constraints) с возможностью хранений NULL-значений;
- расширение состава встроенных функций в соответствии с SQL-стандартом (TRIM, POSITION, ROUND и т. п.);
- поддержка стандартного типа BOOLEAN (принимающего значения TRUE, FALSE или UNKNOWN) в полях, переменных и логических предикатах языка;
- большее соответствие существующих команд SQL-стандарту.

Заключение

Итак, мы совершили обзор событий прошедшего года, которые привели к появлению версии 1.5, и оценили ее возможности. С полной уверенностью можно сделать вывод, что развитие проекта перешло в качественно новую стадию по сравнению с ситуацией, имевшей место один-два года назад. Перечень новых возможностей в языке SQL, а также изменения в архитектуре и сопутствующее улучшение производительности действительно дают новой версии СУБД весьма неплохие шансы на рынке открытых систем. Постоянно растущая инфраструктура проекта, включающая в себя сторонние программные продукты, техническую поддержку и документацию, также способствует укреплению положения проекта в мире Open Source. Все это, включая планы по разработке версии 2.0, позволяет считать Firebird серьезным продуктом, способным быстро и качественно решать задачи, предъявляемые к системам такого класса.

Приложения

Глоссарий

(Автором данного глоссария является Клаудио Р. Вальдеррама (Claudio R. Valderrama).
Публикуется с любезного разрешения автора.
Оригинал находится на сайте www.cvalde.com/document/glossary.htm [9]
Перевод Алексея Ковязина.)

Active tables (активные таблицы). Концепция, применяемая в InterBase для реализации системных таблиц (см. статью ***System tables***). Они активны в том смысле, что изменения в них запускают связанные с этими изменениями логические и физические действия. Например, добавление индекса и его сегментов в соответствующие системные таблицы вызывает распределение страниц, которые будут содержать индексные структуры.

Alternate key (альтернативный ключ). Это уникальный ключ (смотрите ***unique key***), который не является первичным ключом (смотрите ***Primary key***). Такой ключ может использоваться вместо первичного ключа и поэтому называется альтернативным.

BDE. Это аббревиатура от Borland Database Engine. Первоначально созданное как ядро для взаимодействия с базами данных DBASE и Paradox, BDE было расширено набором библиотек, известных как ***Borland SQL Links***, и служит теперь промежуточным программным обеспечением для удаленного подключения к реляционным СУБД через библиотеки SQL Links. Таким образом, BDE поддерживает и навигационный подход и синтаксис SQL. Имя "BDE" используется для того, чтобы обозначить пакет, который содержит технологическое ядро (которое включает в себя инфраструктуру ***IDAPI*** (см. соответствующую статью) и общее ядро для осуществления запросов) плюс три IDAPI драйвера/ядра (для Paradox, dBase и текстовых форматов) плюс механизм ODBC Socket, который превращает любой ODBC-драйвер в IDAPI-драйвер, доступный для использования в IDAPI-приложениях. Важно заметить, что так как BDE связывается с несколькими видами SQL-серверов с помощью SQL, то он не может распознать или воспользоваться преимуществами каждой функции в каждом конкретном реляционном сервере СУБД, поэтому поддержка некоторых функций InterBase в BDE ограничена или вовсе отсутствует.

BLOB. Сокращение для Binary Large Object. InterBase изобрел BLOB-поля несколько лет назад и компания "Боинг" воспользовалась их преимуществами для хранения звукозаписей в базе данных. Насколько я знаю, InterBase предложил первую реализацию BLOB на рынке реляционных СУБД. В InterBase BLOB могут хранить различные подтипы. Однако следует отметить, что изначально слово "BLOB" ничего не означало и, по-видимому, было заимствовано из какого-то кинофильма.

BLR: эта аббревиатура означает Binary Language Representation (двоичное представление языка). Внутри ядра сервера нет ни ***SQL***, ни ***GDML***, ни ***QUEL*** – нет реляционных языков, которые спроектированы, чтобы на них писали и читали люди. Вместо этого запросы даны в двоичном представлении, которое явля-

ется надмножеством известных нам реляционных языков. Утилиты ***GPRE*** и ***QLI*** преобразуют ***SQL*** и ***GDML*** в ***BLR***. Когда вы успешно компилируете хранимую процедуру или триггер, их скомпилированное представление сохраняется в формате BLR в ***BLOB***-полях. Помимо этого, интерфейс ***DSQL*** преобразует все запросы к серверу в ***BLR***. Вы можете воспользоваться утилитой командной строки ***isql*** для того, чтобы просмотреть BLR-представление триггеров, хранимых процедур, ограничений, значений по умолчанию и представлений, для этого необходимо выполнить команду SET BLOB ALL и затем выбрать соответствующие BLR-поля из системных таблиц (см. ***System tables***).

Character set (набор символов). Языки программирования обычно используют не более двух типов наборов символов – ASCII и UNICODE. Первый набор ограничен 256 возможными символами (включая контрольные последовательности), тогда как второй набор содержит 65536 возможных символов. Проблема состоит в том, что в базе данных необходимо хранить символы без чрезмерных затрат и в то же время необходимо корректно передавать эти символы клиентам, которые ожидают увидеть правильное изображение для каждого сохраненного кода, обозначающего символ. Не все кодовые таблицы используют одинаковые числовые значения для одних и тех символов, и к тому же некоторые алфавиты полностью отличаются друг от друга или содержат специальные символы. Таким образом, важно определить набор символов, соответствующий кодовой странице, которую будет использовать клиентское приложение. InterBase поддерживает как возможность задать набор символов для всей базы данных, так и возможность явно указать набор символов для каждого поля типа char или varchar. Не существует набора символов, доступного везде внутри сервера. Если не указывать набор символов, то по умолчанию будет установлен набор символов NONE, который означает, что символы хранятся так, как они есть, и не предпринимается попыток преобразовать эти символы при передаче между клиентом и сервером.

Collation (сопоставление). Это понятие связано с операциями на данном наборе символов. Проще говоря, collation – это сравнение. Collation определяет, как инструкция SORT упорядочивает результаты выборки, как работает функция UPPER и как сравниваются поля в выражениях WHERE и HAVING в предложении SELECT.

DDL. Это часть SQL, которая работает с определениями данных, т. е. она управляет метаданными и потому именуется Data Definition Language (язык определения данных).

DML. Это часть SQL, которая работает с данными и потому именуется Data Manipulation Language (язык манипуляции данных).

DPB. Это Database Parameter Buffer (буфер параметров базы данных), массив символов, используемый для передачи параметров и соответствующих им значений серверу при вызове функций InterBase API. DPB включает следующие данные: первый байт, содержащий его версию, а затем идут наборы байтов для каждого параметра, причем каждый набор содержит сначала 1 байт, определяющий тип параметра, затем байт, содержащий длину данного набора после его типа, и потом собственно совокупность байт, которая несет ту или иную информацию в зависимости от типа параметра.

DSQL. Это аббревиатура для Dynamic SQL (динамический SQL). Когда вы читаете в документации: "доступно в DSQL", это означает, что инструкция доступна в программах, которые посылают на сервер SQL-команды (с параметрами или без), созданные во время выполнения. Этот способ работать с сервером является обычным для приложений на Delphi или BCB.

DSRI. Аббревиатура для Digital Standard Relational Interface (стандартный Цифровой реляционный интерфейс). Используется в Rdb/VMS, Rdb/ELN, а также в некоторых продуктах компании DEC.

DYN. Это еще один язык с байтовым представлением для описания предложений, описывающих данные. Подсистема ***DSQL*** в InterBace передает разобранные ***DDL*** предложения компоненту, который запускает DYN, а затем ***Y-Valve*** (см. соотв. статью), который интерпретирует DYN и прямо изменяет активные системные таблицы (см. ***Active tables***)

ESQL. Embedded SQL (встроенный SQL). Когда вы читаете в документации "доступно в ESQL", это означает, что инструкция доступна в приложениях, в которые встроены статические SQL-команды в формате BLR. См. статью ***GPRE*** для получения дополнительной информации.

FIB. Сокращение для Free InterBase Components (свободные компоненты для InterBase). Эти компоненты созданы Грегори Детцом (Greg Deatz), тем самым человеком, который создал библиотеки FreeUDFLib (Delphi/Windows) и FreeUDFLibC (C/Sun). FIB позволяет из программ на Delphi соединяться с InterBace напрямую, минуя ***BDE***. FIB является основой для ***IBX*** (см. соотв. статью), и с тех пор, как появился ***IBX***, FIB более не развивался.

FIBPlus. Сергей Бузаджи решил, что FIB достаточно хорош для построения на его основе нового пакета и включил туда множество новых функций, таких, как поддержка InterBase 6, например. Кроме этого, изменения коснулись оптимизации работы с памятью и значительного увеличения скорости работы компонентов. В отличие от FIB, FIBPlus также содержит ряд дополнительных компонентов и экспертов для Delphi и BCB, которые упрощают разработку при использовании FIBPlus. Пакет является коммерческим и в настоящий момент поддерживает все версии InterBase, начиная с 4-й, а также клоны InterBase: Firebird и Yaffil.

Foreign key (внешний ключ, FK). Это специальный не уникальный ключ (см. ***non-unique key***), который используется для автоматического обеспечения ссылочной целостности (см. ***Referential integrity***). В InterBase, когда вы создаете FK, соответствующий индекс создается автоматически (и всегда имеет порядок по возрастанию). Поле в мастер-таблице, на которое ссылается внешний ключ, должно иметь первичный ключ (***Primary key***) или уникальный ключ (***Unique key***).

Gbak, gdef, gfix, gpre, gsec, gstat, iblockpr and qli. Эти утилиты описаны в главе "Состав модулей InterBase" (ч. 4). Помимо этого, ***gpre*** and ***qli*** описаны ниже.

GDB. Это расширение, которое по общепринятому соглашению используется для баз данных InterBase. На самом деле InterBase может использовать любое расширение для базы данных. Это расширение служит напоминанием о тех днях, когда компания InterBase Corp называлась Groton Database Systems.

GDDL. Аббревиатура для Groton Data Definition Language. Был инструмент, который анализировал выражения на GDML и генерировал вызовы, которые определяли базы данных. GDML является эквивалентом для ***DDL*** в ***SQL***.

GDML. Аббревиатура для Groton Data Manipulation Language. Этот реляционный язык был создан для использования людьми, как и SQL.GDML – первоначальный язык InterBase до сих пор поддерживается в утилите, называемой ***QLI***. GDML является эквивалентом ***DML*** в ***SQL***, но в то же время в него входят некоторые возможности ***DDL***.

GPRE. Это препроцессор InterBase. В зависимости от платформы он поддерживает несколько языков и позволяет писать на них приложения, использующие встроенный ***SQL***. GPRE на основе используемых в этих приложениях SQL-команд создает команды в ***BLR***-формате. Чаще всего препроцессором поддерживается язык C, но в зависимости от платформы можно найти поддержку для языков COBOL и ADA. Когда вы читаете в документации: "доступно в GPRE", это означает, что описываемая инструкция может быть использована в программах со статическими SQL-командами. Эти программы предварительно обрабатываются препроцессором GPRE и затем передаются компилятору языка (например, C++) для создания исполняемого файла.

Hierarchical database (иерархическая база данных). Первый (изначальный) вид базы данных. Каждый объект (исключая корень) должен иметь одного и только одного родителя. Компания IBM использовала этот вид базы данных до 1970 года.

IBX. Сокращение для InterBase Express. Начиная с Delphi 5 компания Borland решила выпускать два "экспресс"-продукта – ADO Express и IBX. IBX представляет собой пакет компонентов, которые осуществляют прямое соединение с InterBase, используя его API, а не ***BDE***. IBX был создан на основе FIB и был встроен в абстрактную иерархию наборов данных в VCL (Visual Component Library – основная библиотека в Delphi).

IBO. Сокращение для InterBase Objects. IBO – это альтернативный, созданный "с нуля" коммерческий пакет для доступа к InterBase минуя BDE, через InterBase API. Его разработал Джейсон Вартон (Jason Wharton). IBO – предназначенный для работы с базами данных пакет, основанный на оригинальных возможностях InterBase по управлению данными и транзакциями. Разработка IBO основывается на модели клиент-сервер.

IDAPI. Сокращение для Integrated Database API (интегрированный прикладной программный интерфейс для работы с базами данных). IDAPI объединяет навигационный доступ к данных (ISAM) и ориентированный на запросы (SQL или QBE) доступ в модель согласованного курсора (consistent cursor model). IDAPI появилось в середине 1994 года и объединяет в себе работу, проделанную техническим комитетом IDAPI (включающим в себя компании Borland, IBM, Novell и Wordperfect). Первый вариант IDAPI был известен как BDE версии 2 (не существовало BDE версии 1, так как ранее она называлась ***ODAPI***). Термин "IDAPI" используется не только для того, чтобы сослаться на часть BDE, реализующую API, но и для того, чтобы указать всю технологию целиком. Термин "IDAPI" вытеснил термин "ODAPI", который использовался в ранних реализациях этой технологии.

Identity key(идентифицирующий ключ). Хотя не существует строгого определения этого ключа, идентифицирующий ключ обычно представляет собой суррогатный ключ (см. ***Surrogate key***) на основе поля типа INTEGER, которое автоматически заполняется сервером и может предполагать неявное создание лежащего в его основе индекса. Например, и Paradox и MS SqlServer имеют специальный тип поля для этих целей. InterBase явно не поддерживает такие типы полей, однако тот же самый результат может быть достигнут с помощью полей типа INTEGER, генератора и триггера (с большими усилиями – но и с большей гибкостью). Триггер типа BEFORE INSERT перед вставкой записи в таблицу вызывает функцию gen_id, которая получает следующее значение генератора. Приращение значения генератора может быть не равным единице, как в Paradox.

ISC: Сообщения об ошибках в InterBase начинаются с сочетания букв ISC, и поэтому люди часто спрашивают, что такое ISC. Это не более чем InterBase Software Corporation, наполовину независимая компания, принадлежавшая Borland. В 1998 году она была окончательно поглощена Borland. В конечном счете ISC возродилась в виде Open Source-проекта InterBase – Firebird.

Isolation level(уровень изоляции). Это характеристика транзакции, которая определяет, как одна транзакция должна взаимодействовать с другими транзакциями в той же самой базе данных, в терминах видимости и блокировок. Обычно используются 3 уровня изоляции: Dirty Read (грязное чтение), Read Commited (подтвержденное чтение) и Repeatable Read (повторяющееся чтение). Первый режим является единственным уровнем изоляции в Paradox и поддерживается немногими реляционными СУБД. Второй уровень используется по умолчанию в почти всех реляционных серверах. Третий режим используется по умолчанию в InterBase, и реализован он таким образом, чтобы его использование не ставило сервер "на колени" (что не является фактом для других серверов). Помимо перечисленных уровней изоляции, InterBase предлагает еще более высокий уровень, известный как Repeatable Read with Stability (стабильное повторяющееся Чтение), который может использоваться для специальных и очень коротких транзакций. Благодаря возможности хранить множество версий записей InterBase также позволяет явно контролировать управление блокировками (избежать, ждать или выдать ошибку). Подобный контроль не имеет смысла на других "классических" серверах. (См. главу "Транзакции в InterBase. Параметры транзакций". – прим. авт.).

ISQL. Interactive SQL, утилита для интерактивного выполнения команд SQL. Когда вы читаете в документации: "доступно в ISQL", это означает, что описываемой инструкцией можно воспользоваться в инструменте пользователя для того, чтобы послать SQL-команды на сервер и увидеть результаты этих запросов.

JRD. Внутренняя часть (ядро) сервера InterBase. Это ядро обязано своим именем Джиму Старки (Jim Starkey), создателю JRD, которая является основой InterBase. JRD расшифровывается как Jim's Relational Database, что отличает ее от параллельно (и официально) разрабатывающейся части RDB в компании DEC.

Key. Несмотря на тот факт, что одной из характеристик реляционного множества является отсутствие повторяющихся элементов, теория множеств сама по себе не дает ответа, как избежать повторения. В то же время в реляционной теории важно иметь возможность однозначно идентифицировать запись. Кроме то-

го, хотя множество по определению не является упорядоченным, но упорядочение записей является способом избегать их дублирования, улучшать производительность и группировать кортежи. Существуют следующие типы ключей: ***unique (уникальный)***, ***primary (первичный)***, ***alternate (альтернативный)***, ***surrogate (суррогатный)***, ***identity (идентифицирующий)***, ***non-unique (неуникальный)*** и ***foreign (внешний)***.

LIBS. Это сокращение для Local InterBase Server (локальный InterBase-сервер). В прошлом, когда InterBase был закрытым, коммерческим продуктом, было два типа лицензий: локальная (позволяющая устанавливать соединения только с того же компьютера, на котором находится InterBase) и удаленная (позволяющая устанавливать соединения с удаленных компьютеров). Версии, которые поставлялись вместе с Delphi и Borland C++ Builder, были локальными и имели лицензии только для разработки. InterBase 6.0.1 не имеет ни лицензий, ни коммерческих ограничений, и потому понятие LIBS исчезло. Реализация LIBS никогда не отличалась с точки зрения кода от обычного InterBase, всегда использовался тот же самый исходный код, на который налагались ограничения, основывающиеся на лицензиях, которые можно было купить и установить.

MGA. Это аббревиатура для Multi Generational Architecture (архитектура множественных поколений). Это другое наименование многоверсионного ядра сервера, которое позволяет InterBase избегать блокирования и в то же время быстро восстанавливаться в случае отказа (поломки сервера, выключения питания и т. д.) без использования протокола транзакции (см. ***Transaction log***).

Natural scan (натуральный перебор). В случае, если это возможно и имеет смысл, то сервер для поиска записей использует индекс, когда в SQL-предложении появляются части WHERE или GROUP BY. Когда сервер решает пройтись по таблице с первой по последнюю запись в том порядке, в котором они хранятся, то говорят, что используется натуральный перебор. Заметьте, что иногда оптимизатор InterBase прав, когда он решает, что натуральный перебор быстрее, чем использование индекса.

Network database (сетевая база данных). Второй вид баз данных после иерархических. Каждый объект может иметь несколько указателей на другие объекты. Таким образом создается ячейка связей и отсюда происходит имя "сетевая".

Non-unique key (NUK, неуникальный ключ). Это ключ (смотрите статью ***key***), который служит для того, чтобы идентифицировать диапазон записей, так как множество записей могут иметь то же самое значение этого ключа. Такой ключ предназначен для упорядочения записей и выполнения условий ссылочной целостности (см. ***Referential integrity***). В InterBase невозможно создать NUK как часть определения таблицы, вы должны непосредственно создать лежащий в основе NUK индекс. Исключением является случай, когда вы создаете внешний ключ (см. ***Foreign key***) – в этом случае неуникальный ключ создается автоматически.

Null value (неопределенное значение). В действительности это должно называться Null state – неопределенное состояние. По существу NULL – это не значение. Он лишь указывает, что данному полю не было присвоено конкретного значения, поэтому его содержимое неопределенно. NULL – это не нуль и не пустая строка (нулевой длины), потому что все это определенные значения. Операции над NULL возвращают либо NULL, либо UNKNOWN и являются источ-

ником замешательства для начинающих разработчиков реляционных баз данных, тем более что проверка на NULL в запросе производится с использованием оператора IS NULL, а занесение в поле значения NULL – присвоением NULL нужному полю.

Object Oriented database (объектно-ориентированная база данных). Это новый тип баз данных, который все еще находится в разработке. В таких базаах данных действительно хранятся объекты. Эти базы данных не требуют нормализации. Схема базы данных отражает структуры данных в программе, и наоборот. Связи представлены указателями. Идея этих СУБД напоминает сетевые базы данных (см. статью ***network databases***), но имеют более проработанную концепцию, потому что объекты выполняют некоторые задачи по управлению базы данных. Несмотря на множество усилий по расширению рынка для объектно-ориентированных серверов, они остаются в своей нише.

ODAPI. Это означает Open Database API (открытый прикладной интерфейс программирования баз данных). Этот проект начался в 1990 году по инициативе Borland и впервые появился в СУБД Quattro Pro for Windows 1.0 в сентябре 1992 года. Это было начало проекта ***BDE***.

ODBC. Сокращение для Open Database Connectivity. Это интерфейс уровня вызовов (call-level interface), который позволяет приложениям получать доступ к данным в любой базе данных, для которой существует ODBC-драйвер. Используя ODBC, возможно создавать приложения, которые работают с данными в любом базе данных, для которой конечный пользователь имеет ODBC-драйвер. Интерфейс ODBC предоставляет API, которое позволяет приложению не зависеть от сервера базы данных. В настоящее время InterBase v5 имеет ODBC-драйвер от компании Visigenic и более новый, поставляемый компанией Intersolv. К сожалению, ни один из них не предлагает полноценной функциональности, причем оба имеют ошибки и уже не разрабатываются. Новое поколение драйверов для InterBase 6 появилось в конце 2000 года – в том числе драйверы Gemini InterBase и Intersolve for InterBase 6.

ODS. Это аббревиатура для On-Disk Structure (буквально переводится как "структура на диске"). Это внутренняя структура базы данных InterBase. Для InterBase 4.0 ODS имело версию 8, для InterBase 4.2 – 8.2, для InterBase 5.x – 9 и для InterBase 6 – 10.

OLAP. Аббревиатура для On-Line Analytical Processing (оперативная аналитическая обработка). Объемы данных выросли настолько, что никто из людей, принимающих решения, не в состоянии просмотреть все необходимые данные. Менеджерам нужно определять направления, рассматривать исторические перспективы, пробовать различные сочетания типа "если что-то произойдет, то…" и т. д. Для того чтобы получать такие совокупные отчеты, серверы баз данных должны обладать достаточным быстродействием, чтобы прочитать многие мегабайты в пакетном режиме и в то же время клиенту вернуть результат за приемлемое время.

OLEDB. Это стандарт, разработанный компанией Microsoft, которая решила включить термин "OLE" в название некоторых своих технологий. Это низкоуровневая спецификация для доступа к реляционным и нереляционным данным. Предполагается, что эта технология будет более эффективной, чем ODBC, для реляционных баз данных. В 2001 году существовала реализация OLE DB только под Windows. Термин "ADO" обозначает надстройку над OLEDB.

OLTP. Это On-Line Transaction Processing (оперативная обработка транзакций). Это классическое требование к серверу баз данных: клиенты должны совершать транзакции в реальном времени. В большинстве случаев вызываются короткие транзакции, в рамках которых выполняются определенные действия вроде обновления записей о покупателях. Задачи формирования отчетов более продолжительные, но они обычно не меняют данные.

Primary key (первичный ключ, или PK). Это уникальный ключ (см. ***Unique key)***, который используется для уникальной идентификации каждой записи в таблице и выполняет роль внешнего ключа (см. ***Foreign key***) в зависимых таблицах. PK может содержать одно или больше полей. В InterBase во время создания первичного ключа автоматически создается соответствующий индекс – причем всегда в возрастающем порядке.

PSQL. Когда вы читаете в документации: "доступно в PSQL", это означает, что описываемая инструкция может быть использована в хранимых процедурах и триггерах на сервере. Отличия между процедурами и триггерами минимальны: процедуры не могут использовать специальные переменные OLD и NEW, в то время как триггеры не имеют параметров и не могут использовать предложение EXIT. Вы не можете явно вызывать триггеры из DML-выражений или из другого триггера или процедуры.

QLI. Это аббревиатура для Query Language Interpreter (интерпретатор языка запросов). Это интерактивный инструмент для извлечения данных (и манипуляции ими) из баз данных InterBase. QLI поддерживает значительные подмножества языков ***GDML***, ***SQL*** и ***GDEF***.

QUEL. Был такой академический реляционный язык, который предшествовал ***SQL***. СУБД Ingres и Britten-Lee первоначально были QUEL-серверами.

RDB. Легко догадаться, что это аббревиатура для Relational Database (реляционных СУБД). Это была первая попытка компании DEC создать СУБД, используя такой подход. В то же время RDB стала начальным этапом создания InterBase, в котором теперь с префикса RDB$ начинаются все системные таблицы (см. ***System tables***). for documentation purposes.

RDB$DB_KEY. Обычно упоминается как DB_KEY, это одна из недокументированных возможностей InterBase, которая может быть использована в динамическом SQL (см. ***DSQL***) и в хранимых процедурах.

Referential integrity (ссылочная целостность). В теории баз данных это концепция, которая по-простому может быть объяснена следующим образом: если А зависит от Б и производится попытка удалить или изменить Б таким образом, что какой-либо из изменяемых атрибутов нарушит зависимость А от Б, то в этом случае изменяемое действие должно быть либо отклонено, либо А должно быть изменено таким образом, чтобы синхронизироваться с изменениями в Б (т. е. чтобы зависимость не нарушилась); или если Б удаляется, то А также должно быть удалено или изменено таким образом, чтобы зависеть от чего-то другого. Ссылочная целостность реализована как автоматический механизм в сервере баз данных, который позволяет поддерживать ссылочную целостность (правила ссылочных ограничений) так, как это установлено пользователем с помощью настроек, определенных в стандарте SQL. Эти настройки распознаются и реализуются в каждом конкретном сервере СУБД.

Relational database (реляционная СУБД). Тип СУБД, который реализован с таких продуктах, как like InterBase, Oracle и Sybase. Основываясь на сильной теоретической базе, разработанной Коддом (E. F. Codd) и К. Дэйтом (Chris Date), реляционная модель баз данных следует совокупности концепций в математике, в которой взаимосвязи представлены атрибутами "связей". Строго говоря, схема базы данных (структура ее объектов) должна проходить "нормализацию", в течение которой база данных приводится к "нормальным формам", именующимся "первой", "второй", "третьей", "Бойса – Кодда" (Boyce-Codd) и "пятой". На практике база данных, приведенная к 3-й нормальной форме, считается нормализованной. Одной из тем, связанных с нормализацией (которая до сих пор вызывает дискуссии), является представление и использование неопределенных значений (см. ***Null value***).

Special system tables (специальные системные таблицы). Это две системные таблицы (см. ***System tables***), которые находятся вне контекста транзакций. В отличие от остальных пользовательских и системных таблиц изменения в этих таблицах видны любой пользовательской транзакции без необходимости сделать подтверждение (commit) или откат (rollback). Это таблицы **rdb$formats** и **rdb$pages**. Вы можете прочитать в InterBase 6 Language Reference, что содержат данные таблицы. Компиляторы и серверы баз данных – это наиболее типичные случаи кода, которые зависят от своих собственных метаданных, используемых при описании данных и других метаданных, однако этот замкнутый круг должен быть где-то разорван, чтобы не попасть в ловушку "что было раньше: курица или яйцо".

SQL. Стандартный язык для управления реляционными СУБД. SQL – это сокращение для Structured Query Language. SQL является декларативным языком, потому что, будучи преобразованным в процедурные языки типа C или Pascal, SQL определяет вещи, которые должны быть выполнены сервером базы данных в терминах ожидаемых результатов, но не то, КАК они должны быть выполнены. Однако в SQL были добавлены функции для контроля над некоторыми особенностями сервера с помощью явного использования планов. Можно проследить причины добавления этих функций в работах компании IBM в 60-х годах.

[Available in] ESQL, DSQL, PSQL and isql ([доступно в] ESQL, DSQL, PSQL and isq). Если вы читаете руководства по InterBase, то можете увидеть фразы "доступно в SQL, DSQL и isql", когда объясняются какие-либо команды. В данном случае **SQL** означает Embedded SQL (***ESQL***), т. е. команды InterBase, которые можно писать внутри базового языка (C в данном случае), затем пропускать эти программы через препроцессор (***GPRE***), чтобы сгенерировать исходный код на C, который будет использоваться в вашем приложении. Это статические SQL-команды. **DSQL** означает динамический SQL (***DSQL***), т. е. команды SQL, которые можно создавать и отправлять InterBase во время выполнения программы. Такие команды не надо предварительно компилировать перед тем, как запустить приложение. И наконец, **isql** означает Interactive SQL и служит для обозначения инструментов, с помощью которых можно работать с InterBase, набирая команды и просматривая результаты их выполнения. Есть только один "родной" isql инструмент – инструмент командной строки isql.

SQL Links (связи с SQL). BDE может быть расширена дополнительными ***IDAPI*** SQL-драйверами, которые обеспечивают прозрачную возможность соединения с широко используемыми SQL-серверами без применения ODBC. Например, существуют SQL-драйвера для серверов InterBase, Oracle, Sybase, MS SQL и Informix. Эти драйверы называются SQL Links, потому что они связывают BDE с удаленными серверами базы данных.

Surrogate key (суррогатный ключ). Когда ни одно поле или комбинация полей в таблице не могут быть уникальными для каждой записи, то в качестве первичного ключа нужно использовать "искусственный" ключ (см. ***Key***). Обычно это или случайное сгенерированное значение (наподобие GUID), или постоянно возрастающее значение. Суррогатные ключи используются теми разработчиками, которые полагают, что первичные ключи (см. ***Primary key***) должны основываться на генерируемых полях и не должны быть частью естественных атрибутов данных в моделируемой предметной области.

Sweeping. В InterBase это процесс, который собирает и освобождает старые и ненужные версии записей в базе данных. Этот процесс запускается при достижении порогового значения (известного как Sweeping Interval) и является следствием многоверсионной архитектуры InterBase-сервера. В других коммерческих СУБД такого процесса нет. Процесс Sweeping может быть явно вызван с помощью утилит администрирования InterBase. Sweeping – это сборка "мусора", выполняемая для каждой таблицы в базе данных.

System tables (системные таблицы). Реляционные СУБД самодостаточны. Это означает, что данные о структуре пользовательских таблиц также хранятся в таблицах. Эти таблицы, которые хранят данные о данных (метаданные или схему данных), создаются автоматически и называются системными таблицами. Они содержат информацию в том числе и о самих себе, что похоже на попытку выяснить, что был раньше – курица или яйцо. По соглашению названия системных таблиц и их полей начинаются с префикса RDB$. Однако что действительно отличает системные объекты от остальных, так это особый флаг, распознаваемый InterBase'ом, который хранится в специальном поле в системных таблицах, – в этих таблицах хранится информация о различных объектах базы данных (таблицах, процедурах, генераторах и т. д.).

Transaction (транзакция). Логический набор действий, включающий в себя посылку одной или нескольких команд на сервер баз данных. Транзакция является атомарным действием: либо все действия в рамках транзакции выполняются полностью, либо полностью отменяются.

Transaction log (лог транзакций). Обычная реляционная СУБД (и некоторые объектно-ориентированные СУБД) использует отдельный файл, в котором хранится история транзакций. Когда происходит какая-либо поломка, сервер при запуске читает этот файл и определяет, какие изменения в базе данных нужно подтвердить, а какие отменить. InterBase не пользуется такими приспособлениями, потому что в случае возникновения поломки многоверсионная архитектура сервера (см. ***MGA***) позволит начать работу сервера немедленно, а от ненужных версий записей, оставшихся от неподтвержденных транзакций, избавляться при следующей операции чтения-изменения этих данных.

Transaction zero. Все пользовательские транзакции могут только видеть подтвержденные данные или сообщить об ошибке, если новейшая версия записи была создана в рамках другой транзакции, но еще не подтверждена. Но существует транзакция № 0, которая запускается сервером. Эта транзакция запущена в особом состоянии предварительной завершенности, поэтому она может видеть все изменения, произведенные в рамках всех транзакций, и завершенных, и подтвержденных, и все версии записей. Это необходимо, например, для осуществления условий ссылочной целостности (смотрите ***referential integrity***) и для обслуживания индексов (так как индексы отслеживают все версии во всех полях, на которые они распространяются).

UDF. Сокращение для User Defined Function (определенные пользователем функции). В InterBase имеется небольшое количество встроенных функций, которые определяются стандартом SQL. Для расширения функциональности разработчик может писать функции, вызываемые InterBase-сервером таким образом, что они могут использоваться как встроенные. Библиотека FreeUDFLib служит демонстрацией возможностей UDF, также предоставляя набор очень полезных и часто требующихся функций.

Unique key (уникальный ключ). Значение, которое идентифицирует каждую запись (кортеж) в таблице и отличает ее от остальных записей. Следовательно, только одно значение уникального ключа используется для каждой записи. Простейшим типом уникального ключа является поле, значения которого не могут повторяться, как идентификатор (табельный номер) работника внутри компании. Часто одно поле не удовлетворяет условиям уникальности, поэтому нужно использовать комбинацию полей. Если не существует подходящей для уникального ключа комбинации, то используют суррогатный ключ (см. ***Surrogate key***). Когда объявляется уникальный ключ в InterBase, то автоматически создается соответствующий индекс, причем всегда в порядке возрастания.

Y-Valve. Внутри InterBase имеет несколько "серверов", и когда происходит подсоединение, то InterBase должен решить, какой из серверов использовать для конкретной базы данных. Алгоритм для определения нужного сервера называется Y-valve (по определению Стива Тентона (Steve Tendon)). Помимо прочего Y-Valve должен определить, использовать ли прямой доступ к базе данных (локальная база данных) или подсоединиться к удаленному InterBase-серверу (в InterBase Classic-архитектуры) и какую версию использовать для чтения данной ODS (в случае, когда один и тот же InterBase-сервер может читать разные версии ***ODS***).

Дополнение к русскому переводу:

GUID (Global Unique Identifier) – глобальный уникальный идентификатор. Под GUID обычно понимается 128-битовое уникальное значение, которое получается с помощью механизма генерации GUID на основе текущей даты и времени, а также системных номеров процессора и материнской платы. Механизм GUID позволяет получить огромное количество уникальных идентификаторов. Поэтому GUID может быть использован для уникальной идентификации различных объектов (например, COM-серверов).

Параметры конфигурационного файла InterBase

Этот документ был написан Анн В. Харрисон (Ann W. Harrison).
Публикуется с любезного разрешения автора.
Переведено Алексеем Ковязиным, Алексеем Карякиным, 2002 год.
Консультанты – Дмитрий Кузьменко, Алексей Флегонтов.
Оригинал http://www.ibphoenix.com/ibp_config.html)

Внимание! Символ # в ibconfig означает комментарий! Для применения параметора его надо удалить из начала строки!

LOCK_MEM_SIZE

Параметры в ibconfig

V4_LOCK_MEM_SIZE 98304
ANY_LOCK_MEM_SIZE 98304

Действие

LOCK_MEM_SIZE определяет количество памяти, выделяемый для таблицы блокировок. В случае сервера с архитектурой Classic, указываемый размер используется для начального выделения памяти, а затем таблица блокировок может расширяться во время работы, пока не займет всю свободную память. В случае SuperServer устанавливаемый размер невозможно изменить без перезапуска сервера. По умолчанию размер памяти, выделяемой для блокировочной таблицы, равен 98304 байтам.

Объяснение

Во всех версиях InterBase, исключая те, которые исполняются под управлением ОС VMS, конфликты распределения ресурсов разрешаются с помощью таблицы блокировок, которую ведет InterBase (только при употребление VMS InterBase пользуется его менеджером блокировок). В архитектуре Classic таблица блокировок находится в совместно используемой всеми серверными процессами памяти. В архитектуре SuperServer таблица блокировок является частью самого серверного процесса.

Хотя InterBase не употребление блокировки для разрешения конфликтов на уровне записей, он все же использует блокирование для того, чтобы защитить страницу базы данных в момент ее изменения. Под "моментом изменения" имеется в виду не блокировка во время транзакции, а блокировка страницы в момент ее непосредственного изменения, когда какой-либо клиент пишет туда данные. Помимо этого InterBase использует блокировки, чтобы позволить одной транзакции ожидать окончания другой, если возник конфликт, а также в ряде случаев, когда требуется синхронизация.

Показания к изменению параметра

Изменение размера таблицы блокировок может повлиять:

1. на размер кеша страниц базы данных. Каждая страница, помещаемая в кеш, блокируется по крайней мере один раз, а страницы, которые читаются несколькими клиентами, могут блокироваться несколько раз (этот пункт относится только к архитектуре Classic).
2. на число одновременных транзакций. Каждая транзакция имеет блокировку, которая ее идентифицирует. Блокировка используется для синхронизации транзакций, а также для того, чтобы распознать случаи, когда транзакция завершилась без подтверждения (commit) или отката (rollback).
3. на события. Механизм оповещения о событиях основывается на блокировках. Число событий и число клиентов, ожидающих эти события, влияют на размер таблицы блокировок.

SEMAPHORE COUNT

Параметры в ibconfig

V4_LOCK_SEM_COUNT 32
ANY_LOCK_SEM_COUNT 32

Действие

В системах, не поддерживающих многопоточную обработку, этот параметр устанавливает число семафоров, доступных InterBase. Количество семафоров по умолчанию зависит от ОСи описывается в следующей таблице:

Операционная система	Количество семафоров по умолчанию
EPSON SEMAPHORES	10
M88K SEMAPHORES	10
UNIXWARE SEMAPHORES	10
NCR3000 SEMAPHORES	25
SCO_UNIX SEMAPHORES	25
sgi SEMAPHORES	25
IMP SEMAPHORES	25
DELTA SEMAPHORES	25
Ultrix SEMAPHORES	25
DGUX SEMAPHORES	25
DECOSF SEMAPHORES	16
Other UNIX	32

Объяснение

Семафоры используются для блокировок и сообщений о событиях. Теоретически InterBase должен использовать очень маленькое количество семафоров – 32 должно быть более чем достаточно.

Показания к изменению параметра

Если в файле протокола InterBase InterBase.log вы видите сообщение об ошибке "semaphores are exhausted", то следует увеличить количество семафоров.

LOCK SIGNAL

Параметры в ibconfig

V4_LOCK_SIGNAL 16
ANY_LOCK_SIGNAL 16

Действие

Параметр изменяет номер сигнала, используемый для обозначения конфликтов блокировок.

Объяснение

В архитектуре Classic, когда один серверный процесс блокирует страницу базы данных или другой ресурс, который необходим второму процессу, второй процесс сигнализирует об этом первому. Чтобы сменить номер сигнала, используется данный параметр. Значение номера сигнала по умолчанию зависит от ОС:

```
NETWARE_386  BLOCKING_SIGNAL 101
WINDOWS_ONLY BLOCKING_SIGNAL 101
All Others BLOCKING_SIGNAL SIGUSR1
```

Показания к изменению параметра

Сигналы имеют тенденцию "зашумляться", потому что несколько разных служб ОС могут использовать один и тот же сигнал. InterBase спроектирован для работы с "зашумленными" сигналами. Когда он получает сигнал, то пересылает его другим процессам-обработчикам, которые обрабатывают этот сигнал, причем с InterBase ничего не случится, если он получит сигнал и не сумеет его обработать, т. е. InterBase устойчив к "случайным" сигналам-шумам.

Может случиться так, что другой процесс в операционной системе использует тот же сигнал, что и InterBase. Тогда в случае, если этот процесс не сможет передать сигнал или аварийно завершиться при виде сигнала, который он не может обработать, то вы увидите, что либо InterBase-соединение "зависло", либо ошибки возникнут в другом процессе. В этом случае можно использовать параметр LOCK SIGNAL, чтобы выбрать другой сигнал.

Для систем с ОС Windows нет никакой необходимости изменять этот параметр.

EVENT MEMORY SIZE

Параметры в ibconfig

V4_EVENT_MEM_SIZE 32768
ANY_EVENT_MEM_SIZE 32768

Действие

Параметр устанавливает начальный размер памяти, выделенной для таблицы событий (events).

Объяснение

Таблица событий (event table) хранится в отображенной (mapped) памяти. В архитектуре Classic место под эту таблицу выделяется для каждого клиентского соединения. В архитектуре SuperServer одна таблица совместно используется всеми клиентами.

Показания к изменению параметра

Таблица увеличивается динамически, поэтому вроде бы нет причины для того, чтобы устанавливать этот параметр.

DATABASE CACHE SIZE

Параметры в ibconfig

DATABASE_CACHE_PAGES 75

Действие

Этот параметр устанавливает число страниц из любой базы данных, которое может одновременно находиться в кеше. Если вы увеличиваете это значение, InterBase поместит больше страниц из каждой базы данных в кеш. По умолчанию SuperServer помещает в кеш 2048 страниц из каждой базы данных, а Classic – 75 страниц на каждое клиентское соединение. На 16-битовых версиях Windows по умолчанию размер кеша 50 страниц.

Объяснение

Кеш содержит страницы, которые были прочитаны из базы данных, а также вновь созданные страницы. Назначение кеша – уменьшить число чтений-записей страниц в базе данных путем удержания их в ОЗУ, чтобы они были "под рукой", пока подтверждение транзакции (commit) или другое событие не вынудит их быть записанными. Чем больше кеш, тем больше страниц сохранеются в памяти.

Минимальное значение кеша – 50 страниц, и максимальное – 65535. Эмпирический опыт показывает, что значения кеша более 10000 уменьшают производи-

тельность. По информации компании Borland, проблема снижения производительности при кеше размером более 10000 буферов ликвидирована в InterBase 6.5.

Вы можете увеличить размер кеша на уровне базы данных (в SuperServer) или на уровне соединения клиент-база данных путем использования параметра соединения, который можно использовать в ISQL, в Server Manager, в IBConsole.

InterBase не увеличивает размера кеша динамически, потому что слишком большой кеш может быть таким же вредным, как и слишком маленький. Например, массовая вставка записей работает лучше при использовании маленького кеша, потому что страницы, которые наполняются данными, не посещаются вновь. Те приложения, которые применяют часто используемые справочные страницы, могут применять больший кеш для того, чтобы хранить эти таблицы в памяти.

Показания к изменению параметра

Если вам кажется, что ваш InterBase сервер-работает слишком медленно и число страниц в кеше менее 10000, то увеличение размера кеша может улучшить производительность.

SERVER PRIORITY CLASS

Параметры в ibconfig

SERVER_PRIORITY_CLASS 1

Действие

Устанавливает приоритет для SuperServer на Windows/NT/2000. Значение 2 этого параметра устанавливает высокий приоритет (HIGH_PRIORITY_CLASS) серверному процессу InterBase – ibserver.exe. Все остальные значения будут устанавливать серверному процессу InterBase значения нормального приоритета (NORMAL_PRIORITY_CLASS). По умолчанию параметр имеет значение 1.

Объяснение

Увеличивая приоритет процесса, вы можете заставить InterBase-сервер занимать больше процессорного времени. Если вас заботит производительность, вам лучше поставить сервер на выделенную однопроцессорную систему. Если вы рассчитываете на прирост производительности от клиентов, запущенных на одной машине, в том случае, когда они применяют совместную память, а не TCP, используйте многопроцессорную систему, привяжите сервер к одному процессору, а клиентов запускайте на другом.

Показания к изменению параметра

Я несколько предубеждена против этого параметра, но если хотите попробовать, то вперед.

SERVER CLIENT MAPPING

Параметры в ibconfig

SERVER_CLIENT_MAPPING 4096

Действие

Этот параметр устанавливает размер области разделяемой памяти, которая используется в Windows-системах для того, чтобы устанавливать связь между сервером и клиентом, запущенным на той же машине (локальное соединение). Размер по умолчанию – 4 Кбайт.

Объяснение

На Windows-системах (и только Windows) клиент, запущенный на той же машине, что и сервер, может устанавливать соединение с сервером через область разделяемой памяти, а не через TCP/IP. Используйте этот параметр для управления размером этой области.

Память выделяется блоками по 1024 байта. Приемлемый диапазон значений лежит между 1-м и 16-м однокилобайтовым блоком, т. е. значение этого параметра может быть одним из следующих: 1024, 2048, 3072, 4096, 5120, 6144, 7165, 8192, 9216, 10240, 11264, 12288, 13312, 14336, 15360 или 16384.

Показания к изменению параметра

Если у вас много памяти и локальных клиентов, то увеличение размеров области обмена (communications area) может улучшить производительность.

SERVER WORKING SIZE

Параметры в ibconfig

SERVER_WORKING_SIZE_MIN 0
SERVER_WORKING_SIZE_MAX 0

Действие

Этот параметр устанавливает ограничения размера рабочей физической памяти (working size), доступно SuperServer на платформе Windows/NT/2000. Параметр измеряется в однокилобайтовых блоках. По умолчанию оба параметра имеют значение 0, что означает "нет ограничений".

Объяснение

Ограничивая максимальный размер рабочей памяти, можно заставить InterBase "упасть замертво" раньше времени из-за недостатка памяти. Увеличивая минимальный размер рабочей памяти, вы можете заставить InterBase "захватывать" память тогда, когда она ему не нужна.

Показания к изменению параметра

Установка минимального размера рабочей памяти может устранить некоторые затраты на постепенное разрастание памяти сервера, т. е. выделить столько памяти, чтобы серверу не пришлось больше ее увеличивать и тратить на это какие-то усилия. Установка максимального размера рабочей памяти может удержать сервер от захватывания всей доступной памяти на системах с малым ее количеством. Не запускайте InterBase SuperServer на системах с малым количеством памяти.

LOCK GRANT ORDER

Параметры в ibconfig

V4_LOCK_GRANT_ORDER 1

Действие

Устанавливает состояние блокировки 1 – "Истина", включает сортировку блокировок; 0 – "Ложь", и выключает режим сортировки блокировок. По умолчанию сортировка блокировок выключена.

Объяснение

Сортировка блокировок достаточно проста, необходимо только узнать немного больше о блокировках. Когда соединение (клиент) запрашивает блокировку на объект, оно указывает в запросе определенный уровень блокировки. Типы блокировок приведены в следующей таблице:

Идентификатор типа блокировки	Английское наименование	Русский перевод наименования блокировки
#define LCK_none 0		Отсутствие блокировок
#define LCK_null 1	Existence	Блокировка существования объекта
#define LCK_SR 2	Shared Read	Совместное чтение
#define LCK_PR 3	Protected Read	Защищенное Чтение
#define LCK_SW 4	Shared Write	Совместная запись
#define LCK_PW 5	Protected Write	Защищенная запись
#define LCK_EX 6	Exclusive	Эксклюзивная блокировка

Блокировка типа LCK_none на самом деле представляет собой запрос на снятие существующей блокировки. Блокировка LCK_null – это блокировка существования, которая налагается клиентским соединением. Для этого соединения важно лишь, чтобы заблокированный объект существовал.

Этот тип блокировки используется для того, чтобы гарантировать существование индексов, пока существуют скомпилированные запросы, которые зависят от этих индексов. Взаимодействие уровней блокировки описывается в следующей таблице совместимости блокировок (здесь 1 означает, что данные блокировки совместимы, 0 – несовместимы):

	none	null	Shared Read	Protected Read	Shared Write	Protected Write	Exclusive
none	1	1	1	1	1	1	1
null	1	1	1	1	1	1	1
SR	1	1	1	1	1	1	0
PR	1	1	1	1	0	0	0
SW	1	1	1	0	1	0	0
PW	1	1	1	0	0	0	0
EX	1	1	0	0	0	0	0

Когда соединение желает заблокировать объект, оно подает запрос на блокировку, который определяет блокируемый объект и желаемый уровень блокировки.

Обычно если объект, который какое-то соединение желает заблокировать, уже блокирован другим соединением, то выстраивается очередь доступа, причем соединения, которые желают получить уровень блокировок ниже, чем тот, что уже наложен на объект, продвигаются вперед очереди! То есть если объект был заблокирован с уровнем Protected Read, то следующие соединения, которые запрашивают на этот объект блокировку Protected Read или ниже, передвинутся в начало очереди (точнее, пройдут без очереди), а соединения, которые запрашивают блокировки уровнем выше (например, Shared Write), будут "топтаться" в очереди. Если загрузка базы данных очень велика, такое поведение может привести к тому, что соединения, которые требуют высоких уровней блокировок, могут ждать неопределенно долго, потому что новые читающие соединения (которые запрашивают низкие уровни блокировки) будут постоянно прибывать и "лезть без очереди".

Показания к изменению параметра

Если ваши операции по записи данных имеют более низкий приоритет по сравнению с операциями чтения, включение сортировки блокировки улучшит производительность чтения, особенно в течение длинных читающих транзакций. Но необходимо помнить, что даже только читающие транзакции изменяют базу данных, – по крайней мере они записывают информацию о состоянии своих транзакций.

LOCK HASH SLOTS

Параметр lock hash slots был удален из конфигурационного файла InterBase6.x, по крайней мере в SuperServer под NT. Однако исходный код для того, чтобы прочитать и интерпретировать этот параметр, все еще существует.

Параметры в ibconfig

LOCK_HASH_SLOTS 101

Действие

Этот параметр определяет ширину хэш-таблицы, которая используется для поиска блокировок. По умолчанию значение этого параметра 101. Число должно быть простым, чтобы хэш-алгоритм производил хорошее распределение. Он может быть в диапазоне от 101 до 2048.

Объяснение

Представьте себе, что хэш-таблица – это одномерный массив с цепочками, которые "свисают" из каждой ячейки этого массива. Менеджер блокировок хэширует имя объекта и затем вычисляет остаток от целочисленного деления этой величины на число хэш-слотов в массиве, таким образом он определяет ячейку, на которую надо "подвесить" блокировку данного объекта.

Когда менеджер блокировок ищет определенную блокировку, он определяет ячейку хэш-массива аналогичным образом, а затем спускается вниз по цепочке, подвешенной к данной ячейке, и ищет объект с правильным именем. Если находится более одного объекта с этим именем, он проходит по цепочке "однофамильцев", которая подвешивается к первому объекту, который соответствует искомому имени.

Что получается в итоге? Чем длиннее будут цепочки, подвешенные к каждому слоту, тем медленнее будет работать менеджер блокировок. В среднем каждая цепочка должна иметь не более 10 ячеек.

Показания к изменению параметра

Первым признаком для изменения этого параметра должна быть общая низкая производительность системы с большим количеством пользователей и страниц в кеше. Запустите инструмент iblockpr из директории %INTERBASE%\Bin для печати блокировок. Если средняя длина более 10, увеличьте число хэш-слотов для этого параметра. Для начала умножьте среднюю длину цепочек на текущее число слотов и поделите на 9, а затем возьмите простое целое число, большее полученного значения (но меньшее 2048). Если вы производите подобную настройку на SuperServer, то необходимо также увеличить размер таблицы блокировок.

DEADLOCK TIMEOUT

Параметры в ibconfig

DEADLOCK_TIMEOUT 10

Действие

Этот параметр определяет число секунд, в течение которых менеджер блокировок будет ожидать разрешения обнаруженного конфликта, а по истечении этого срока конфликт будет рассмотрен как потенциальный deadlock (взаимная блокировка).

Объяснение

Применение этого параметра интенсивно тестировали около двух лет назад на системах, которые были такими медленными, что сегодня они не могли бы работать даже в посудомоечной машине. На данное время 10 с являются оптимальным интервалом. Если установить меньший интервал, то проверки на deadlock перегрузят компьютеры, а если более длинный, то пользователи ворвутся в вашу лабораторию и разгромят компьютеры.

Показания к изменению параметра

Deadlock очень редко встречаются в InterBase. Обычная ошибка deadlock, ошибка обновления (Update Conflict) не является тем deadlock, который обнаруживается менеджером блокировок. Представляет интерес запрограммировать действительный случай возникновения deadlock (когда A изменяет строку 1, B изменяет строку 2, затем A пытается изменить строку 2, а B – строку 1, причем все это без подтверждения изменений – без commit), а затем поэкспериментировать с различными интервалами DEADLOCK_TIMEOUT, чтобы увидеть, как изменяется производительность.

LOCK ACQUIRE SPINS

Параметры в ibconfig

LOCK_ACQUIRE_SPINS 0

Действие

Для архитектуры SuperServer этот параметр не производит никакого действия. В архитектуре Classic только один клиент одновременно может обращаться к таблице блокировок. Доступ к таблице блокировок определяется мьютексом. Запрос мьютекса может быть либо условным, когда задержка является ошибкой и запрос должен быть повторен, либо безусловным, когда запрос будет ожидать до тех пор, пока его не обслужат. Параметр LOCK_ACQUIRE_SPINS устанавливает число попыток выполнения условного запроса. По умолчанию – 0 (нуль).

Объяснение

Условный запрос мьютекса повторяется определенное число раз, устанавливаемое параметром LOCK_ACQUIRE_SPINS, а затем превращается в безусловный запрос. Вроде бы это может принести пользу на машинах с несколькими процессорами (SMP). Что весьма сомнительно.

Показания к изменению параметра

Нет.

CONNECTION TIMEOUT

Параметры в ibconfig

CONNECTION_TIMEOUT 180

Действие

Устанавливает время ожидания (тайм-аут) соединения. По умолчанию – 180 с.

Объяснение

Чтобы распознать клиентов, которые некорректно разорвали соединение, включая тех Windows-клиентов, которые выключили свои компьютеры не закрыв приложения, InterBase посылает фиктивный пакет в течение времени ожидания (тай-маута) соединения. Если ответа на запрос нет в течение установленного времени, то InterBase разрывает соединение.

Время ожидания также может быть указано в dpb (database parameter block). Соответствующий параметр имеет название isc_dpb_connect_timeout.

Показания к изменению параметра

Чем выше значение этого параметра, тем меньше фиктивных запросов будут загружать сеть. С другой стороны, "мертвые" соединения будут дольше "висеть". Рекомендуется значительно увеличить значение этого параметра, если вы точно уверены, что клиентские приложения не будут некорректно завершать свою работу.

DUMMY PACKET INTERVAL

Параметры в ibconfig

DUMMY_PACKET_INTERVAL 60

Действие

Этот параметр определяет, насколько часто будут посылаться фиктивные запросы для проверки того, что клиент все еще работает. По умолчанию это 60 с.

Объяснение

InterBase закрывает соединение, когда клиент перестает отвечать. Для того чтобы определить, что клиент более не отвечает на запросы, InterBase ожидает некоторое время (определяемое параметром CONNECTION_TIMEOUT), а затем посылает фиктивный запрос для проверки соединения. Если при посылке возникает ошибка, то InterBase заключает, что клиент "мертв".

Вы можете настроить частоту, с которой посылаются фиктивные пакеты, либо с помощью этого конфигурационного параметра, либо на уровне соединения, установив в структуре dpb параметр isc_dpb_dummy_packet_interval.

Показания к изменению параметра

Чем выше это значение, тем реже фиктивные пакеты будут появляться с сети. Но, с другой стороны, "мертвые" соединения будут дольше "висеть". Рекомендуется значительно увеличить значение этого параметра, если вы точно уверены, что клиентские приложения не будут некорректно завершать свою работу.

Примечание

Есть непроверенная информация, что значение 0 отключает посылку фиктивных пакетов.

TMP DIRECTORY

Параметры в ibconfig

TMP_DIRECTORY <size> <quoted directory string>
TMP_DIRECTORY 20000 "/opt/InterBase/tmp"

Действие

Этот параметр может использоваться в файле ibconfig несколько раз для того, чтобы определить местоположение временных файлов InterBase. Размер временных файлов задается в байтах. Если в ibconfig нет этого параметра, то InterBase проверяет следующие переменные окружения: INTERBASE_TMP, TMP, и TEMP.

Этот параметр доступен только для архитектуры SuperServer.

Объяснение

InterBase использует временные файлы для разнообразных операций, особенно для хранения промежуточных результатов сортировки. Этот конфигурационный параметр позволяет вам определять список каталогов, которые будут использоваться для хранения временных файлов. Повторение параметра позволяет добавлять дополнительные каталоги.

Показания к изменению параметра

Установка этого параметра позволяет назначить ряд временных каталогов и точно определить количество места, которое будет использовано в каждом из них.

EXTERNAL FUNCTION DIRECTORY

Параметры в ibconfig

EXTERNAL_FUNCTION_DIRECTORY <quoted directory string>
EXTERNAL_FUNCTION_DIRECTORY "/opt/InterBase/my_functions"

Действие

Этот параметр может быть использован в ibconfig несколько раз, для того чтобы назначить местоположение для библиотек пользовательских функций (UDF). Если этот параметр отсутствует, то InterBase проверяет каталоги INTERBASE/UDF или $INTERBASE/intl. Этот параметр доступен только для варианта SuperServer.

Объяснение

InterBase ищет в заданных этим параметром каталогах библиотеки, которые он загружает по ссылке. Этот параметр позволяет задать любое число каталогов, в которых InterBase будет искать библиотеки пользовательских функций (UDF) или определения наборов символов (character set definitions).

Показания к изменению параметра

Если необходимо использовать большее число каталогов или другие каталоги, отличные от стандартных, то используйте этот параметр.

TCP REMOTE BUFFER

Параметры в ibconfig

TCP_REMOTE_BUFFER 8192

Действие

Этот параметр устанавливает максимальный размер пакетов TCP, используемых при обмене между клиентом и сервером. Возможны значения от 1448 до 32768 байт.

Объяснение

InterBase производит упреждающее чтение для клиента и может посылать несколько строк данных в одном пакете. Чем больше размер пакета, тем больше данных пересылается за один раз.

Показания к изменению параметра

Сильно загруженная сеть.

Примечание

В конфигурационном файле InterBase все параметры по умолчанию закомментированы с помощью символа # в начале строки. Если вы изменяете значение какого-либо параметра, не забудьте убрать символ комментария!

Инструменты администратора и разработчика InterBase

В комплекте с InterBase 6.x и его клонов, например Firebird, обычно поставляется инструмент IBConsole. Мы не рекомендуем его использовать – вместо него существует множество других инструментов администратора и разработчика InterBase. Ниже приводится список наиболее известных инструментов, краткое их описание и условия приобретения. Рекомендуем попробовать все эти инструменты для того, чтобы найти наиболее оптимальный вариант для своих задач.

Универсальные инструменты администратора и разработчика InterBase

IBExpert

www.ibexpert.com
Универсальный инструмент для администрирования и разработки, русифицирован. Поддерживает plug-ins. Очень популярный инструмент.
Для xSSR – бесплатно, для остальных – 199 евро.

Devrace™ BlazeTop ™

www.blazetop.com
Универсальный инструмент для администрирования и разработки, русифицирован.
Для xUSSR – бесплатен. Включает русскую документацию.

IBWorkbench

www.InterBaseworkbench.com
Универсальный инструмент для администрирования и разработки, нерусифицирован.
Коммерческий продукт, есть триал-версия

IBAccess

http://sourceforge.net/projects/ibaccess
Универсальный инструмент для администрирования и разработки баз данных InterBase, есть версии как под ОС Windows, так и под Linux, нерусифицирован.
Бесплатно.

IBAdmin

www.sqlly.com/ibadmin2.htm
Универсальный инструмент для администрирования и разработки, нерусифицирован.
Коммерческий продукт, есть триал-версия.

Marathon

http://www.gimbal.com.au/

Универсальный инструмент для администрирования и разработки. Один из наиболее известных интсрументов.

Бесплатно.

IBManager

http://www.ibmanager.com

Универсальный инструмент для администрирования и разработки, нерусифицирован. Существует в двух различных редакциях: Lite и Professional.

Коммерческий продукт.

Специализированные инструменты

MiTeC InterBase Performance Monitor

ftp://ftp.ibphoenix.com/downloads/mitecibpm.zip

Инструмент для сбора статистики сервера и базы данных. Интегрируется с приложениями на FIBPlus. Нерусифицирован.

Бесплатно.

IBSurgeon Viewer 1.0.2

www.ibsurgeon.com

Уникальный инструмент для исследования внутренней структуры баз данных InterBase/Firebird и диагностики их повреждений.

Бесплатно.

IBSurgeon IBFirstAID 1.0

www.ibsurgeon.com www.ib-aid.com

Инструмент для автоматического восстановления тяжело поврежденных баз данных. Позволяет лечить до 70% повреждений баз данных. Разработан на базе библиотеки прямого доступа к базе данных (минуя сервер InterBase/Firebird). Диагностическая версия – бесплатная, полная версия - коммерческий продукт.

IBSurgeon IBAnalyst 1.5

www.ibsurgeon.com www.ibanalyst.com

Инструмент для автоматического анализа статистики базы данных InterBase/Firebird, с встроенной экспертной системой оценки состояния и БД и выдачи рекомендаций по эксплуатации и разработке баз данных.

Пробная версия – бесплатная, полная версия - коммерческий продукт

IBAffinity

www.upscene.com/files/ib_affinity.zip
Утилита для привязки серверного процесса InterBase архитектуры SuperServer к одному процессору.
Бесплатно.

IBDatabase Comparer

www.clevercomponents.com
Утилита для сравнения двух баз данных InterBase. Позволяет находить различия в структуре баз данных и формировать на их основе SQL-скрипт для нивелирования различий. Очень полезный инструмент для задачи сопровождения продуктов с множеством установок и версий.
Бесплатно.

IB DataPump

www. clevercomponents.com
Мощный инструмент для переноса данных из одной базы данных InterBase в другую. Также позволяет работать с источниками данных ODBC и BDE. Отслеживает зависимости между объектами базы данных, вычисляет правильный порядок переноса таблиц, позволяет отключать триггеры и т. д.
Бесплатно, но оплачивается поддержка.

GBAK Scheduler

www.ibase.ru/download/gbakschd.zip
Утилита для организации резервного копирования баз данных InterBase. Позволяет задавать расписание создания копий, вести протоколы, сжимать файлы резервных копий с помощью zip.
Бесплатно.

IBReplicator

www.synectics.co.za/borland/InterBase/ibreplicator
Инструмент для организации репликации баз данных InterBase.
Коммерческий продукт.

Grant Manager

www.eadsoft.com/russian/grantmanager/index.htm
Инструмент для управления правами пользователей в базах данных InterBase.
Условно-бесплатная программа.

Посмотреть самый "свежий" список программного обеспечения для разработчиков и администраторов InterBase можно на сайте поддержки данной книги www.InterBase-world.com, а также на сайтах www.ibase.ru и www.ibphoenix.com.

Литература

1. InterBase 6 Data Definition Guide (www.ibase.ru/v6/doc/DataDef.zip)
2. InterBase 6 Language Reference (www.ibase.ru/v6/doc/Langref.zip)
3. InterBase 6 Getting Started (www.ibase.ru/v6/doc/OpGuid.zip)
4. InterBase 6 Operations Guide
5. Diagnosing and Repairing InterBase Database Corruption, by Paul Beach, (http://www.ibphoenix.com/ibp_db_corr.html)
6. InterBase Public License (http://www.borland.com/devsupport/InterBase/opensource/IPL.html)
7. InterBase 6 API Guide (http://www.ibase.ru/v6/doc/ApiGuide.zip)
8. Ivan Prenosil InterBase-related site (www.volny.cz/iprenosil/InterBase/index.htm)
9. Glossary Claudio Valderrama (www.cvalde.com/document/glossary.htm)
10. "The On-Disk Structure of InterBase" by Ann.W.Harrison (http://www.ibphoenix.com/ibp_ods.html)
11. "Space Management in InterBase" by Ann W. Harrison (http://www.ibphoenix.com/ibp_spam.html)
12. "Structure of a Data Page" by Paul Beach (With thanks to Dave Schnepper and Deej Bredenberg) (http://www.ibphoenix.com/ibp_ods_page.html)
13. "Extending SQL Functionality in InterBase" by Charlie Caro, (http://community.borland.com/article/0,1410,27563,00.html)
14. «Введение в SQL», Мартин Грабер, Издательство «Лори», 382 стр.
15. InterBase 5.5 Embedded Installation Guide, Borland
16. InterBase Installation & Deployment Options By David R. Robinson. (http://community.borland.com/article/0,1410,27574,00.html)
17. Как научиться писать UDF для IB Database за 21 мин., Олег Кукарцев, (http://ibase.ru/devinfo/udf_ok.htm)
18. Transaction options explanation by Claudio Valderrama (www.cvalde.com/document/TransactionOptions.htm)

Оглавление

ИЗДАТЕЛЬСТВО **«ОЦ КУДИЦ-ОБРАЗ»**

Тел.: (095) 333-82-11; ok@kudits.ru,
http://www.kudits.ru/publish

КНИГИ В ПРОДАЖЕ

Баженова И. Ю. Delphi 7. Самоучитель программиста. 448 с.

ISBN 5-93378-072-4 Оптовая цена 110 руб.

Эта книга посвящена описанию новой версии одной из наиболее популярных систем разработки приложений Delphi 7.

В книге последовательно изложены концепции объектно-ориентированного программирования, приведен необходимый справочный материал по языку программирования Object Pascal, описана интегрированная среда проектирования IDE и приемы работы с проектами, подробно рассмотрены вопросы программирования графического интерфейса пользователя, применение библиотеки визуальных компонентов VCL и объектов ActiveX, рассмотрены вопросы реализации межсетевого взаимодействия и создания приложений для Internet.

Особое внимание уделено механизмам доступа к серверам баз данных через ODBC и OLE DB.

Книга хорошо иллюстрирована и содержит большой объем справочной информации. Изложение материала сопровождается полезными примерами программ.

Книга предназначена как для разработчиков программного обеспечения, так и для широкого круга пользователей, желающих самостоятельно научиться проектировать приложения в среде Windows.

Баженова И. Ю. C++ & Visual Studio.NET. Самоучитель программиста. 448 с.

ISBN 5-93378-072-4

Эта книга посвящена описанию новой версии одной из наиболее популярных систем разработки приложений Visual Studio.NET.

В книге последовательно изложены концепции объектно-ориентированного программирования, приведен необходимый справочный материал по языку программирования C++ и библиотеке MFC, подробно рассмотрены вопросы программирования графического интерфейса пользователя. В книге обсуждается архитектура документ-отображение и создание на ее основе SDI и MDI приложений. Рассматриваются вопросы программирования ISAPI-расширений для Интернет.

Особое внимание уделено механизмам доступа к серверам баз данных через ODBC и OLE DB.

Книга хорошо иллюстрирована и содержит большой объем справочной информации. Изложение материала сопровождается полезными примерами программ.

Книга предназначена как для разработчиков программного обеспечения, так и для широкого круга пользователей, желающих самостоятельно научиться проектировать приложения в среде Windows.

Вилльямиал Б. Программирование Access 2002 в примерах. 496 с.

ISBN 5-93378-059-6 Оптовая цена 171,6 руб.

Эта книга позволяет читателю не просто быстро начать писать программы с помощью интегрированной среды разработки MS Access 2002 но и правильно разрабатывать свои приложения баз данных, используя всю мощь MS Access 2002.

Она научит читателя фундаментальным основам, таким как нормализация баз данных, создание запросов, работа с объектами и оптимизация, а также создание пользовательских форм и отчетов с помощью программных средств. Эти умения позволят читателю добиться большей эффективности, производительности и профессионального владения средствами Access VBA. Главы книги включают не только полезные советы, примечания, предупреждения и ссылки, но также содержат резюме, которые дают место каждой главы в общей перспективе.

Климова Л. М. Pascal 7.0. Основы практического программирования. Решение типовых задач. 528 с.

ISBN 5-93378-033-2 Оптовая цена 90,2 руб.

Книга написана на основе отработанной методики и лекционного материала, который использовался в процессе обучения студентов в течение ряда лет. Она содержит структурированное лаконичное описание средств языка и основных приемов работы в среде Borland Pascal, сопровождающихся большим количеством примеров.

Характерной особенностью книги является четкая систематизация рассматриваемых вопросов, широкое использование средств структурного программирования, в том числе схем алгоритмов, и графического представления взаимосвязи указателей и адресуемых ими значений.

Особое внимание уделено использованию динамических переменных и указателей. Например, для динамического формирования больших массивов различных типов с помощью массивов указателей и для формирования массивов указателей на подпрограммы.

Книга предназначена для студентов экономических и технических специальностей, школьников, преподавателей, разработчиков программного обеспечения и инженеров, желающих в полной мере освоить и применять возможности языка Pascal 7.0. Она может быть использована также для самостоятельного изучения программирования на языке Pascal 7.0.

Для удобства работы дополнительно к книге можно приобрести дискету, которая содержит полный исходный код всех программных примеров книги, отлаженных в среде Borland Pascal.

Пирумян В. Платформа программирования J2ME для портативных устройств. 352 с.

ISBN 5-93378-060-X Оптовая цена 132 руб.

В этой книге рассказывается, как разрабатывать и создавать приложения Java для платформы J2ME. Платформа Java 2 Micro Edition (J2ME) определяет среду, которая поддерживает Java на так называемых портативных устройствах, таких как компьютерные приставки к телевизору, и персональных мобильных устройствах, таких как карманные компьютеры, мобильные телефоны и пейджеры. В свете скачкообразного роста количества беспроводных компьютерных устройств практическая полезность этой книги очевидна.

Проффит Б. Windows XP Professional. 416 с.

ISBN 5-93378-055-3 Оптовая цена 112,2 руб.

Книга содержит полное описание последней операционной системы, выпущенной Microsoft - Windows XP Professional, созданной специально для корпоративных и профессиональных пользователей.

Это удобное руководство, с помощью которого читатель быстро узнает, как извлечь преимущества из новых свойств Windows XP по поддержке аппаратного и программного обеспечения. Написанная в формате решения конкретных задач, книга специально приспособлена к нуждам корпоративных и профессиональных пользователей и содержит десятки практических решений. Лучший способ познания - практика, и эта книга поможет читателям выполнять задачи, с которыми они будут сталкиваться каждый день.

О'Коннэл Ф. Как успешно руководить проектами. Серебрянная пуля. 288 с.

ISBN 5-93378-050-2 Оптовая цена 121 руб.

Считалось, и считается до сих пор, что проекты в среде высоких технологий, таких, как разработка программного обеспечения или информационные технологии, очень трудно планировать и выполнять, и что доказательство этого – низкие показатели успешности таких проектов.

Эта книга продемонстрирует, что, хотя такие проекты имеют свои собственные уникальные характеристики, нет причин, по которым ими нельзя управлять с высокой степенью уверенности в успешном исходе. Она содержит множество полезных методик, практических советов и рекомендаций, а также вводный учебный курс по применению Microsoft Project для управления проектами.

Тэллес М., Хсих Ю. Наука отладки. 560 с.

ISBN 5-93378-059-6 Оптовая цена 187 руб.

Ошибки в программах неизбежны. Настоящая книга признает этот факт и учит образу мышления, позволяющему гарантированно отыскивать и устранять ошибки. Она нацелена на то, чтобы сделать отладку менее загадочной, более быстрой и эффективной, экипируя читателя знаниями и методиками, необходимыми для оперативной идентификации, отслеживания и устранения ошибок в программах. Этим книга не ограничивается и идет дальше, предлагая практические советы по минимизации ошибок и улучшению их распознаваемости в тех случаях, когда они все же случаются.

Афанасьев Д., Баричев С. Шагни в Internet самостоятельено. Изд. 2-е, переработанное. 224 с.

ISBN 5-93378-026-X Оптовая цена 42 руб.

Афанасьев Д., Баричев С., Плотников О. Office XP. 356 с.

ISBN 5-93378-043-X Оптовая цена 84,7 руб.

Баженова И. Ю. Jbuilder 5. Программирование на Java. 448 с.

ISBN 5-93378-024-3 Оптовая цена 180,4 руб.

Барсуков В. С. Безопасность: технологии, средства, услуги. 496 с.

ISBN 5-93378-017-0 Оптовая цена 99 руб.

Варфоломеев В. И., Воробьев С. Н. Принятие управленческих решений. 288 с.

ISBN 5-93378-019-7 Оптовая цена 99 руб.

Вовк Е. Т., Баричев С. Г., Плотников О. А. Самоучитель работы на компьютере. Изд. 4-е, дополненное. 352 с.
ISBN 5-93378-066-9 Оптовая цена 72,6 руб.

Вовк Е. Т. PageMaker 6.5/7.0. Самоучитель. 352 с.
ISBN 5-93378-032-4 Оптовая цена 121 руб.

Вовк Е. Т. QuarkXPress 5.0. Самоучитель. 288 с.
ISBN 5-93378-058-8 Оптовая цена 88 руб.

Гусев А. А., Ильина Л. В. Программирование в среде 1С: Бухгалтерия. 352 с.
ISBN 5-93378-027-8 Оптовая цена 121 руб.

Елизаветина Т. М. Делопроизводство на компьютере. Изд. 2-е. 304 с.
ISBN 5-93378-044-8 Оптовая цена 72,6 руб.

Елизаветина Т. М. Компьютерные презентации: от риторики до слайд-шоу. 240 с.
ISBN 5-93378-049-9 Оптовая цена 77 руб.

Зензин О. С., Иванов М. А. AES – стандарт криптографической защиты. Конечные поля. 176 с.
ISBN 5-93378-047-2 Оптовая цена 55 руб.

Иванов М. А. Ассемблер в задачах защиты информации. 320 с.
ISBN 5-93378-038-3 Оптовая цена 96,8 руб.

Иванов М. А. Вступительные экзамены по математике в гимназии, лицеи и колледжи. 204 с.
ISBN 5-93378-001-4 Оптовая цена 28,6 руб.

Климова Л. М. СИ++. Практическое программирование. Решение типовых задач. 596 с.
ISBN 5-93378-020-0 Оптовая цена 125,40 руб.

Король В. И. Visual Basic.NET, Visual Basic 6.0, Visual Basic for Applications 6.0. 496 с.
ISBN 5-93378-048-0 Оптовая цена 143 руб.

Куприянова Г. И. Кадровое делопроизводство на компьютере. 256 с.
ISBN 5-93378-014-6 Оптовая цена 50,6 руб.

Мартынов Н. Н., Иванов А. П. Вычисления, визуализация и программирование в среде MATLAB 5.x. 336 с.
ISBN 5-93378-011-1 Оптовая цена 66 руб.

Мартынов Н. Н. Введение в MATLAB 6. 352 с.
ISBN 5-93378-039-1 Оптовая цена 96,8 руб.

Мельников Д. А. Информационные процессы в компьютерных сетях. Изд. 2-е. 256 с.

ISBN 5-93378-002-2 Оптовая цена 72,6 руб.

Пегано Дж. Лечение псориаза – естественный путь. 288 с.

ISBN 5-93378-025-1 Оптовая цена 132 руб.

Померанц О. Ядро Linux. Программирование модулей. 112 с.

ISBN 5-93378-002-2 Оптовая цена 23,1 руб.

Ражиков М. Ю. 1С: практика настройки оперативного учета. 256 с.

ISBN 5-93378-052-9 Оптовая цена 92,4 руб.

Россоловсский А. В. AutoCAD 2002/2002 LT/2000. Справочник команд. 720 с.

ISBN 5-93378-053-7 Оптовая цена 220 руб

Смирнов Н. Java 2 Enterprise. 240 с.

ISBN 5-93378-037-5 Оптовая цена 112,2 руб.

Стотлемайер Д. Тестирование Web-приложений. 240 с.

ISBN 5-93378-064-2 Оптовая цена 94,6 руб

Титов С. ArchiCAD 7.0. Справочник с примерами. 400 с.

ISBN 5-93378-035-9 Оптовая цена 154 руб.

Титов С. ArchiCAD: Полезные рецепты. 272 с + CD.

ISBN 5-93378-030-8 Оптовая цена 171,6 руб.

Ульрих Л. Photoshop для Web-дизайна. 384 с.

ISBN 5-93378-054-5 Оптовая цена 121 руб.

Форд А., Теори Т. Практическая отладка в C++.144 с.

ISBN 5-93378-040-5 Оптовая цена 44 руб.

Форд Дж. Л. Персональная защита от хакеров. Руководство для начинающих. 272 с.

ISBN 5-93378-041-3 Оптовая цена 92,4 руб.

Черкасский В. Т. Эффективная анимация во Flash. 432 с.

ISBN 5-93378-029-4 Оптовая цена 102 руб.

Щербо В. К. Стандарты вычислительных сетей. Взаимосвязи сетей. Справочник. 276 с.

ISBN 5-93378-012-X Оптовая цена 90,2 руб.

ЗАКАЗ КНИГ НАЛОЖЕННЫМ ПЛАТЕЖОМ

Издательство «ОЦ КУДИЦ-ОБРАЗ» осуществляет рассылку книг по почте.
Заказы принимаются по адресу: 121354, Москва, а/я 18; или по e-mail: ok@kudits.ru
Условия рассылки: Сумма наложенного платежа складывается из оптовой цены книг, накладных расходов «ОЦ КУДИЦ-ОБРАЗ» на пересылку (30% от стоимости книг) и почтовых расходов (по тарифам почты РФ).

Заказы из регионов России с авиадоставкой, а также заказы из стран ближнего и дальнего зарубежья обслуживаются только по предварительной оплате.

ИЗДАТЕЛЬСТВО
«ОЦ КУДИЦ-ОБРАЗ»

Тел.: (095) 333-82-11; ok@kudits.ru,
http://www.kudits.ru/publish

ПРИОБРЕТАЙТЕ КНИГИ У НАШИХ ПАРТНЕРОВ

Барнаул
Социалистический пр-т, 117а,
(3852) 38-18-72

Великий Новгород
ул. Б. Санкт-Петербургская, 44
тел. (81622) 73-188 доб. 34

Донецк
Книжный рынок "Маяк",
магазин "Специальная литература",
тел. (062) 381-92-32
E-Mail: karimov@skif.net
http:// www.kniga.dn.ua

Екатеринбург
"Книжный мир",
ул. 8 Марта, 8г, (3432) 71-18-87

Краснодар
"Мир книги", ул. Буденного, 147

Москва
"ОПТИМА+"
Розничная торговля
м. Тульская, Варшавское ш., 9,
книжная ярмарка "Центральная",
зеленая линия, павильон 515-49
Оптовая торговля
(095) 333-65-67

Нижний Новгород
"Нижегородский Дом книги",
ул. Советская, 14,
тел. (8312) 44-22-73

Новосибирск
"Книжный пассаж",
ул. Ленина, 10а, (3832) 29-50-30
"Сибирский Дом Книги",
Красный пр-т, 153, (3832) 26-62-39
"Книжный мир", пр-т К.Маркса, 51

Киев
"Техническая Книга"
тел. (044) 269-52-23
oleg@irina-press.kiev.ua
www.tk.com.ua

Пермь
ООО "Образование",
ул. Солдатова, 37,
тел. (3422) 45-96-55
тел./факс. (3422) 42-81-16

Омск
"Книжный Мир",
ул. Ленина, 17/19, (3812) 24-32-54

Ростов-на-Дону
"Мир книги", Ворошиловский пр-т, 33,
(8632) 62-54-61

Санкт-Петербург
"Санкт-Петербургский Дом книги"
Невский проспект, 28, тел. (812) 312-01-84
издательство "Наука и Техника"
пр. Обуховской Обороны, 107,
тел. (812) 567-70-25, 567-70-26

Саратов
"Книжный Мир",
пр-т Кирова, 32, (8452) 32-98-14

Ставрополь
"Книжный Мир", ул. Мира, 337, (8652) 35-47-90

Томск
"Книжный Мир", ул. Ленина, 141, (3822) 51-07-16

Уфа
ООО ПКП "Азия", тел./факс: (3472) 50-39-00
Оптовая торговля
Ул. Зенцова, 70
Розничная торговля
Магазин "Оазис", ул. Чернышевского, 88
Магазин "Книжник", пр. Октября, 106

Ханты-Мансийск
Магазин "Книги", ул. Ленина, 39

Челябинск
"Книжный Мир", ул. Кирова, 90, (3512) 33-19-58

Ярославль
Магазин "Наука",
ул. Володарского, 63, (0852) 25-95-04

РОЗНИЧНАЯ ТОРГОВЛЯ ПО ИЗДАТЕЛЬСКИМ ЦЕНАМ В МОСКВЕ

Варшавское ш., 9, м. Тульская, трамвай 3, 38, 47 до ост. "Стройдвор"

книжная ярмарка "Центральная", зеленая линия, павильон 515-49