

Работа с компонентами IBX, или использование InterBase eXpress в приложениях на Delphi и C++Builder, с СУБД Firebird и InterBase

 ibase.ru/ibx

KDV, www.ibase.ru, 23.02.2005, обновления – 20.05.2005, [23.05.2005](#), [24.05.2005](#), [26.05.2005](#) (1, 2, 3), [16.10.2005](#), [06.04.2006](#), [30.08.2006](#), [15.02.2007](#), [09.03.2007](#), [28.03.2008](#), [01.09.2008](#), [10.06.2010](#), [10.05.2011](#), [27.01.2012](#), [21.01.2013](#), [04.09.2014](#).

Введение

В Delphi 3 в иерархию компонент работы с данными была заложена возможность написания собственных компонент DataSet. То есть, возможность обращаться к некоему источнику данных, минуя BDE. До этого момента компонент TDataSource мог брать данные только из стандартного DataSet (TTable, TQuery), и для компонент "прямого" доступа приходилось писать собственный TDataSource и компоненты визуализации данных (DBGrid, DBEdit и т. п.) – примером таких компонент является IBObjects.

Но в Delphi 3 ситуация изменилось, и в поставку был включен пример работы с текстовым файлом – (он включен в поставку Delphi до сих пор) Demos/DB/TextData. В результате стали появляться наборы компонент, которые позволяли работать напрямую с любыми источниками данных, в том числе с SQL-серверами. Первым набором таких компонент для InterBase был FreeIBComponents.

FreeIBComponents состоял всего из четырех компонент – TFIBDatabase, TFIBTransaction, TFIBDataSet и TFIBQuery. Дополнительный, пятый, компонент TFIBSQLMonitor служил для мониторинга запросов, отсылаемых через TFIBDataSet и TFIBQuery.

Перед выходом InterBase 6, примерно в конце 1999 года, FreeIBComponents прекратил свое существование, поскольку был передан в Borland для разработки компонент IBX (InterBase eXpress), которые должны были быть включены в поставку Delphi и C++Builder. Одновременно исходные тексты FreeIBComponents были взяты за основу для создания библиотеки FIBPlus Сергеем Бузаджи.

Исходный набор компонент был существенно расширен как в IBX так и в FIBPlus, и в этой статье будут рассмотрены компоненты IBX, поставляющиеся в Delphi 7.

Внимание! С помощью IBX (и FIBPlus) можно работать с любыми версиями InterBase, Firebird и Yaffil. Это означает, что все версии IBX и FIBPlus работают (и будут продолжать работать) со всеми версиями InterBase, Firebird и Yaffil, во всех

средах разработки Delphi (3, 4, 5, 6, 7, 2005, 2006, 2007, 2009, 2010, XE, XE2, XE3, XE4), и C++Builder (версий 1, 2, 3, 4, 5, 6, 2006, 2007, 2009, 2010, XE, XE2, XE3, XE4).

В Delphi 2009, 2010, XE-XE8, 10, поставляется новая версия IBX, которая имеет несовместимости с базами данных Firebird в UTF8. По работе с Unicode в Delphi 2009 с IBX и InterBase/Firebird читайте [FAQ](#).

Если Вы используете "чистую" установку Firebird, в которой клиентская gds32.dll называется иначе (fbclient.dll) – воспользуйтесь утилитой instclient.exe из комплекта Firebird.

Обновления

Для IBX нужно обязательно устанавливать обновления, которые можно скачать либо [с оригинальной страницы](#), либо [здесь](#).

Компоненты

Закладка InterBase на палитре компонент в Delphi и C++Builder. Есть в любой версии Delphi/C++Builder (кроме бесплатного Turbo Explorer)



Практически каждый компонент имеет свой собственный редактор свойств, который вызывается по двойному клику на него (если он находится на форме или в DataModule), или в отдельном пункте меню, вызываемому по правой кнопке мыши.

Клиентская библиотека

IBX может работать только с библиотекой gds32.dll. Для работы с Firebird необходимо при помощи входящей в комплект утилиты instclient создать gds32.dll из fbclient.dll, т. к. компоненты ориентируются на версию gds32.dll не ниже 6.0, а в fbclient.dll указана версия Firebird, которая ниже 6.0 (1.5, 2.0, 2.1, 2.5).

Usage:

```
instclient i[nstall] [ -f[orce] ] library  
q[query] library  
r[emove] library
```

Наиболее удобным является размещение gds32.dll рядом с вашим exe. В этом случае приложение загрузит именно эту библиотеку, а не какую то другую, например, из System32.

Также нужно помнить, что компоненты в IDE тоже загружают gds32.dll. Поэтому перед началом работы необходимо проверить диски на присутствие лишних, старых или "неправильных" gds32.dll.

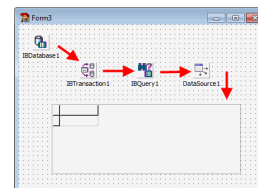
Также см. "Установка только клиентской части".

Организация доступа к данным

Исключительно для начинающих.

Взаимосвязи компонент выглядят следующим образом:

1. первым является IBDatabase. Это центральный компонент для соединения с базой данных. Один компонент может быть связан только с одной базой данных в конкретный момент времени.
2. вторым является IBTransaction. Вне контекста транзакции в InterBase и Firebird нельзя выполнить никаких действий с данными и метаданными БД.



`IBTransaction1.DefaultDatabase:=IBDatabase1;`

3. третьим является либо датасет (IBDataSet, IBQuery), либо IBSQL. он связывается с базой данных и транзакцией

`IBQuery1.Database:=IBDatabase1;`

`IBQuery1.Transaction:=IBTransaction1;`

4. четвертый – источник данных для датасета, т. е. универсальный TDataSource.

`DataSource1.DataSet:=IBQuery1;`

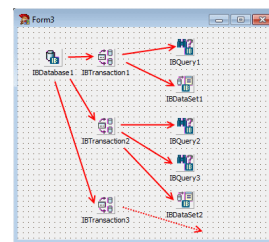
5. пятым является DBGrid. Он связывается только с DataSource.

`DBGrid.DataSource:=DataSource1;`

Легче всего данные связи осуществить прямо в дизайнера, в свойствах компонент, помещая их по очереди (в указанном порядке) на форму. То же самое (установку взаимосвязей) можно выполнить в коде, в аналогичной последовательности.

Для вывода данных в грид требуется заполнить IBQuery1.SQL запросом на выборку (например, select * from employee), а затем

1. IBDatabase1.Open;
2. IBTransaction1.StartTransaction;
3. IBQuery1.Open;



О том, какие свойства IBDatabase, IBTransaction и т. п. необходимо менять и настраивать, читайте дальше.

Необходимо отметить, что в InterBase и Firebird является нормальным

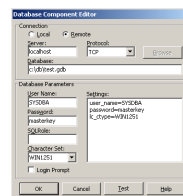
- привязка нескольких IBTransaction к одному IBDatabase. При этом операции в разных транзакциях можно совершать по очереди без проблем.
- привязка нескольких датасетов (IBDataSet, IBQuery, IBSQL и т. п.) к одной транзакции. С датасетами в одной транзакции также можно работать по очереди без проблем.

TIBDatabase

TIBDatabase предназначен для осуществления соединения с базой данных.

Основные свойства:

- **DatabaseName** – имя сервера и путь к базе данных (или алиас, если поддерживается сервером). Например, localhost:c:\dir\data.gdb, server:c:\dir\data.gdb. При ошибках локального соединения (путь к БД без имени сервера) см. [пункт FAQ](#).
- **Params** – параметры соединения: имя пользователя, пароль, чарсет и т. п.



Если воспользоваться редактором свойств TIBDatabase (двойной клик на компоненте), то упомянутые свойства будут заполнены автоматически. Например, для базы данных, созданной с default character set win1251 параметры будут такими:

```
user_name=SYSDBA  
password=masterkey  
lc_ctype=WIN1251
```

На диалоге редактора свойств есть кнопка Test, которая позволяет проверить соединение с указанными сервером и базой данных. Если проверка прошла нормально, то можно установить свойство Connected в True.

Для задания свойств остальных компонент ([TIBTable](#), [TIBQuery](#), [TIBDatabase](#)) потребуется соединение с сервером БД при помощи вышеупомянутого Connected:=True.

В параметрах TIBDatabase можно дополнительно прописывать разнообразные параметры (константы) соединения, указанные в [ApiGuide.pdf](#) (isc_dpb...). Обработка этих констант производится процедурой GenerateDPB в модуле IBDatabase.pas.

Дополнительные свойства:

- **Connected** – управление подсоединением к БД, или проверка состояния соединения
- **DefaultTransaction** – компонент [IBTransaction](#), который будет использоваться в качестве умолчательного для выполнения различных операций IBDatabase. Если это свойство не назначено явно, IBDatabase создаст себе экземпляр IBTransaction самостоятельно.
- **IdleTimer** – по умолчанию 0. Определяет время, в течение которого при отсутствии активных действий соединение с БД закроется автоматически.

- **SQLDialect** – 1 или 3. По умолчанию 3. Определяет диалект, в котором будет работать клиентская часть. Собственно, когда IBDatabase подсоединяется к серверу и БД, из БД считывается номер диалекта, и устанавливается SQLDialect. Менять не рекомендуется.
- **TraceFlags** – перечень действий между клиентом и сервером, которые будет отслеживать компонент IBSQLMonitor или внешние приложения, поддерживающие такую функциональность.

Подсоединение к БД

Соединиться с базой можно установив один раз параметры при помощи вышеописанного диалогового окна. Если требуется заполнить параметры соединения вручную, в том числе вызывая **свой собственный диалог запроса имени пользователя и пароля**, то можно использовать следующий код:

```
IBDatabase1.Params.Clear;
IBDatabase1.LoginPrompt:=False;
IBDatabase1.DatabaseName:='servername:c:\dir\data.gdb';
IBDatabase1.Params.Add('user_name=SYSDBA');
IBDatabase1.Params.Add('password=masterkey');
IBDatabase1.Params.Add('lc_ctype=win1251');
IBDatabase1.Connected:=True;
```

Замена диалога запроса имени пользователя и пароля

Выше уже было сказано, что можно использовать свой диалог для запроса имени пользователя и пароля. Его можно "вставить" в качестве замены стандартного, при LoginPrompt:=True. Пример:

```
procedure TForm1.IBDatabase1Login(Database: TIBDatabase; LoginParams: TStrings);
```

```
var
```

```
dlg: TDBLoginDialog; // созданный вами диалог
```

```
begin
```

```
dlg:=TDBLoginDialog.Create(Application);
```

```
if dlg.ShowModal = mrOK then
```

```
with LoginParams do
```

```
begin
```

```
Values['USER_NAME'] := User_Name;
```

```
Values['PASSWORD'] := User_Pass;
```

```
// другие параметры, например role_name, lc_ctype и т. д., если необходимо
```

```
end;
```

```
dlg.Free;
```

```
end;
```

Динамическое создание диалога не обязательно, но желательно, чтобы он зря не занимал в приложении память, тем более что диалог этот обычно вызывается 1 раз при старте приложения.

Создание БД

Обычно базу данных создают в IBConsole, IBExpert, IBDevStudio, isql и вообще любом инструменте разработчика. Из приложения БД создавать обычно не принято, т.к. в этом случае потребуется все метаданные (структуры таблиц, триггеров и т.п.) создавать опять же в приложении, что увеличит размер приложения. Гораздо проще вместе с приложением поставлять пустую или наполненную частично базу данных. Однако, создать БД из приложения не проблема. Вот как это можно сделать

```
IBDatabase1.Params.Clear;  
IBDatabase1.DatabaseName:='servername:c:\dir\data.gdb';  
IBDatabase1.Params.Add('user "SYSDBA" password "masterkey" ');  
IBDatabase1.Params.Add('page_size 4096');  
IBDatabase1.Params.Add('default character set win1251');  
IBDatabase1.CreateDatabase;  
IBDatabase1.Connected:=False;
```

После чего можно подсоединиться к БД любым способом, указанным выше. Последняя строка в этом примере (Connected:=False) нужна потому, что в IBX после CreateDatabase база остается в открытом состоянии, но не задан чарсет подсоединения к БД, а также неполноценно инициализировано соединение с БД в IBX. Именно поэтому после создания БД к ней нужно снова подсоединиться нормальным способом.

Внимание! При создании БД разные версии серверов InterBase и Firebird могут по разному устанавливать параметр БД Forced Writes. Если Forced Writes = Off, то в случае сбоя компьютера содержимое базы данных может быть повреждено. В статье изложено, как установить правильно ForcedWrites, как избежать повреждений БД и как ее отремонтировать в случае сбоя.

При использовании IBDatabase (IBX) в многопоточных (multithreaded) приложениях, в том числе с web- или com-серверами, нужно соблюдать следующие правила:

1. Соединение с БД не должно быть "локальным". То есть не c:\dir\data.gdb, а сетевым – localhost:c:\dir\data.gdb.
2. Если при использовании локального соединения возникает ошибка – читайте FAQ.
3. В одном thread допускается работа только с одним IBDatabase.

- Попытка осуществить работу с одним коннектом из разных threads может быть успешной, если использовать блокировки thread при обращении к этому коннекту (на мютексах, семафорах и т. п.). Но в результате работа всего приложения будет не многопоточной, то есть, превратится в псевдо-многопоточное по причине блокировок между threads при работе с одним коннектом. Действительно параллельно в разных thread могут выполняться только операции над разными коннектами (IBDatatase).
- Иногда при подсоединении к БД в созданном thread может возникнуть ошибка на вызове `isc_attach_database` (собственно на функции, которая и осуществляет соединение к БД при вызове `IBDatatase.Connected:=True`). В этом случае вынесите открытие соединения в главный thread приложения, а дальнейшие операции с коннектом производите в пределах нужного thread.

TIBTransaction

TIBTransaction – компонент для явной работы с транзакциями.

Клиентская часть InterBase допускает выполнение любых действий только в контексте транзакции. Поэтому если вы смогли получить доступ к данным без явного вызова `TIBTransaction.StartTransaction`, то значит где то в недрах IBX этот вызов произошел автоматически. Такое поведение крайне не рекомендуется использовать. Для корректной работы приложений с базой данных желательно управлять транзакциями вручную, то есть явно вызывать методы `StartTransaction`, `Commit` и `Rollback` компонента `TIBTransaction`.

Основные свойства:

Active – управление стартом или завершением транзакции, а также проверка состояния транзакции.

Внимание! При вызове `Active:=False` транзакция будет безусловно завершена по `Rollback`! Тип завершения транзакции, установленный в `DefaultAction`, в этом случае не имеет значения.

- **DefaultDatabase** – компонент `TIBDatabase`
- **Params** – параметры транзакции
- **AllowAutoStart** – если `False`, то любые попытки автоматического старта транзакций будут пресекаться. По умолчанию `True`.
- **IdleTimer** – если не 0, то время, через которое транзакция будет завершена по `DefaultAction`

- **DefaultAction** – результат автоматического завершения транзакции в случае окончания IdleTimer – taCommit, taRollback, taCommitretaining, taRollbackretaining. Также именно этим способом будут завершены все открытые транзакции в момент вызова IBDatabase.Close (Connected:=False).

Умолчательное значение зависит от версии компонент, и может быть как taRollback так и taCommit.

AutoStopAction – свойство, аналогичное DefaultAction по значениям (плюс saNone, по умолчанию), указывает на метод завершения транзакции, когда все DataSet-ы, подключенные к ней, закрываются.



В параметрах транзакции (картинка слева, вызывается по двойному клику на компоненте, или при нажатии кнопки ... в свойстве Params) указываются константы, соответствующие IB API (без префикса isc_tpb_). Если параметры не указаны (пусто), то IBX использует параметры транзакции по умолчанию, которые соответствуют уровню изолированности snapshot+wait+write (FIBPlus, в отличие от IBX, при пустых параметрах самостоятельно прописывает параметры "read write read_committed no wait", т. е. с другим уровнем изолированности и режимом ожидания). То есть, в транзакции с такими параметрами не будут видны никакие изменения базы данных (кроме тех, которые выполнены в этой транзакции). Это является самой известной проблемой для начинающих разработчиков, когда "приложение видит изменения только после перезапуска" (и именно поэтому разработчики FIBPlus при обнаружении незадаанных параметров у rFIBTransaction меняют уровень изолированности на read_committed). Для обычной работы (показ данных в гриде и т. п.) обычно используются параметры ReadCommitted (как на рисунке), т. к. они позволяют транзакции видеть чужие, committed изменения базы данных (подробно все возможные параметры описаны [здесь](#)) просто путем повторного выполнения запросов (перечитывания данных).

В приложении можно использовать сколько угодно компонентов TIBTransaction. Худшими случаями являются как один компонент на все приложение (пример ibmastapp из Delphi), так и по компоненту на каждый выполняемый запрос. Вы должны определить сами, сколько вам нужно экземпляров TIBTransaction, в соответствии с задачами вашего приложения.

Внимание! В коде не рекомендуется для старта или завершения транзакции использовать property Active. Рекомендуем явный вызов методов StartTransaction, Commit и Rollback - это отлично самодокументирует приложение, может быть найдено поиском, и кроме того, Active:=False завершит транзакцию безусловным вызовом Rollback.

Для работы в режиме "только чтение" (например, со справочниками или при использовании с "двухтранзакционными" датасетами, т. е. отдельными транзакциями для чтения и записи) в FB 1.x и IB 7.x рекомендуется использовать параметры

- read
- read_committed
- rec_version

Такая транзакция в InterBase 6.0 и выше (все версии IB 7.0, 7.1, 7.5, Firebird и Yaffil) может быть открытой сколь угодно долгое время (дни, недели, месяцы), без блокирования других транзакций или влияния на накопление мусора в базе данных (потому что на самом деле на сервере такая транзакция стартует как committed). Однако, такая транзакция (read_committed) во время перечитывания данных будет видеть все новые committed-изменения, и например, для отчетов, ее использовать нельзя. Более подробно о параметрах транзакций см. документ.

Подробнее об управлении транзакциями см. раздел "Использование и управление IBTransaction в приложениях".

Датасеты

Работать с данными в IBX можно при помощи компонент IBDataSet, IBQuery, IBTable, IBStoredProc, IBSQL, но IBSQL не является датасетом, а IBStoredProc хоть и является датасетом, получать из него выборки невозможно (см. далее).

TIBDataSet, TIBQuery, IBTable, TIBStoredProc унаследованы от TIBCustomDataSet.

IBDataSet

```
TIBDataSet = class(TIBCustomDataSet)
```

IBDataSet – основной компонент, пришедший из FreeIBComponents. Остальные, то есть IBTable, IBQuery (+IBUpdateSQL) – всего лишь модификации, либо чуть расширенные, либо усеченные. Возможностей этого компонента хватает практически для любых целей.

Назначение компонента – буферизация записей, выбираемых оператором SELECT, для представления этих данных в Grid, а также для обеспечения "редактируемости" записи (текущей в буфере (гриде)) путем автоматического или ручного задания запросов Insert, Delete и Update.

Внимание! Для выполнения отдельных запросов insert, update, delete, execute procedure и операторов DDL вместо IBDataSet следует использовать IBQuery или IBSQL.

Основные свойства:

- **BufferChunks** – размер буфера (число записей), аллокируемого IBDataSet-ом за 1 раз.
- **Database** – связь с компонентом IBDatabase
- **DataSource** – ссылка на Master-источник данных (TDataSource) для IBDataSet, используемого в качестве Detail.
- **ForcedRefresh** – при False перечитывание данных (текущей записи) производится только при явном вызове Refresh. При True – происходит автоматически при Post. *Вызов метода Refresh будет перечитывать только одну, текущую запись. Для перечитывания всего набора нужно закрыть и открыть датасет, то есть повторно выполнить запрос, хранимый в свойстве SelectSQL (далее).*
- **GeneratorField** – по нажатию на кнопке ... выводится диалог для конфигурирования "автоинкремента" – какому столбцу, какой генератор, с каким шагом, и по какому действию (onNewRecord, onPost, onServer). Это автоматический вызов инкремента значения генератора и помещения его в столбец, вместо ручного использования генераторов.
- **ParamCheck** – проверять или нет наличие в запросе параметров (:param). По умолчанию True. Для выполнения операторов DDL (create procedure, alter, drop и т.п.) нужно выставить в False.
- **Transaction** – к какой транзакции привязан компонент.
- **UniDirectonal** – при False все выбираемые записи кэшируются в буфере IBDataSet, что позволяет после выборки всех записей перемещаться от начала до конца выборки и обратно не обращаясь к серверу. При True размер буфера ограничен, просмотр записей "вверх" на определенном этапе будет невозможен.
- **UpdateObject** – свойство для подключения IBUpdateSQL, не нужно.
- Запросы:
 - **SelectSQL** – основной запрос, который возвращает данные.
 - **RefreshSQL** – запрос *для обновления текущей строки*. Должен содержать условие отбора по первичному ключу или подобное, для выборки одной записи.
 - **InsertSQL** – запрос для вставки записи
 - **UpdateSQL** – запрос для обновления записи
 - **DeleteSQL** – запрос для удаления записи

Для активации IBDataSet необходимо выполнить ряд действий:

1. Соединить его с нужным IBDatabase, и с IBTransaction (если это не тот IBTransaction, который указан для IBDatabase.DefaultTransaction).
2. Определить запрос, выбирающий данные. В этот момент подсоединенные IBDatabase и IBTransaction должны быть активны. Запрос можно указать

- щелкнув на кнопку ... свойства SelectSQL
- нажав на компоненте правую кнопку, выбрать в меню Edit SQL

Если редактирования данных, возвращаемых запросом SelectSQL, не предполагается, то на этом можно остановиться – после Active:=True компонент выберет данные (которые будут видны, если к IBDataSet уже подсоединены TDataSource и TDBGrid). Иначе можно задать запросы Insert/Update/Delete/Refresh. Это можно сделать вручную (прописав запрос в окне, выводимом по кнопке ... у конкретного свойства), или автоматически.

Нажмите на IBDataSet правую кнопку, выберите DataSet Editor. Появится диалоговое окно. Базовая информация (имя таблицы, столбцы), уже получена из Select SQL. В диалоге нужно указать столбцы первичного ключа (key fields) и обновляемые столбцы (update fields). Как правило, столбцы первичного ключа не обновляются, поэтому их исключают из Update Fields. Теперь можно нажать Generate SQL. Появится диалоговое окно с переключателями 4-х запросов, сгенерированными автоматически.



- **RefreshSQL** – как было сказано выше, для считывания (пересчитывания) одной записи (текущей в гриде, например).
- **InsertSQL** – для вставки записи. Если столбцы первичного ключа не указаны в Update Fields, то они не попадут и в сгенерированный оператор SQL. То есть, нужно отредактировать этот запрос, добавив столбцы первичного ключа в оператор Insert (если только столбцы первичного ключа не заполняются на сервере, что делается редко, и также делает невозможным пересчитывание этих данных после вставки).
- **DeleteSQL** – для удаления текущей записи. Условия отбора в where должны совпадать с RefreshSQL.
- **UpdateSQL** – для обновления текущей записи. Если столбцы первичного ключа не исключены из Update Fields, то они будут упомянуты в update set field =:field, и соответственно могут обновляться. Изменение значений столбцов первичных ключей – нехорошая операция, поэтому даже если на самом деле эти столбцы не меняются, их упоминание надо убрать из set данного оператора (но разумеется, оставить в where).

У IBDataSet есть специальные параметры, которые могут быть использованы в запросах. Параметры именуются автоматически, и соответствуют именам столбцов, предваряемые префиксами old_ и new_. Например, old_client_name, new_client_name. В old_-столбцах хранятся данные текущего столбца до модификации записи, а в new_ – те, которые ввел пользователь при

редактировании.

Простой пример, допустим, есть таблица

```
create table X(
```

```
id int not null,
```

```
name varchar(30),
```

```
constraint PK_X primary key (id))
```

тогда

```
SelectSQL = select * from X
```

```
InsertSQL = insert into X (id, name) values (:ID, :NAME)
```

```
ModifySQL = update X set NAME = :NAME where ID = :OLD_ID
```

```
DeleteSQL = delete from X where ID = :ID
```

```
RefreshSQL = Select ID, NAME from X where ID = :ID
```

Обратите внимание, что автоматически генерируемый UpdateSQL проверяет только соответствие первичного ключа (where field = :old_field). То есть, обновление записи произойдет независимо от старых или новых значений столбцов. Это поведение отличается от умолчательного поведения компонент BDE TTable и TQuery. У них было свойство UpdateMode, которое отвечало за проверки модификации записи. Это свойство позволяло контролировать возможное "перекрытие" обновлений для случаев, когда пользователь выполняет редактирование записи "долго", а другой пользователь может успеть отредактировать эту же запись и сохранить ее раньше. То есть, первый пользователь на этапе редактирования даже не будет знать, что запись уже изменилась, возможно не один раз, и сумеет "затереть" эти обновления своим:

upWhereAll (по умолчанию) – проверка на существование записи по первичному ключу + проверка всех столбцов на старые значения. Например,

```
update table set ... where pkfield = :old_pkfield and client_name = :old_client_name and  
info = :old_info ...
```

То есть, в данном случае запрос поменяет информацию в записи только в том случае, если запись до нас никто не успел изменить. Особенно это важно, если существуют взаимозависимости между значениями столбцов – например, минимальная и максимальная зарплата, и т. п.

upWhereChanged – проверка записи на существование по первичному ключу + плюс проверка на старые значения только изменяемых столбцов.

```
update table set ... where pkfield = :old_pkfield and client_name = :old_client
```

В этом случае мы меняем имя клиента со старого на новое. Менялись ли остальные столбцы, пока мы редактировали запись, нас не интересует.

upWhereKeyOnly – проверка записи на существование по первичному ключу.

Последняя проверка соответствует генерируемому автоматически для UpdateSQL запросу. Поэтому, при возможных конфликтах обновлений в многопользовательской среде необходимо дописывать условия к where самостоятельно. И, разумеется, также необходимо при реализации аналога upWhereChanged удалять лишние изменения столбцов в update table set ... – то есть, оставлять в перечне set только действительно измененные столбцы, иначе запрос переписшет чужие обновления этой записи.

Как вы понимаете, это означает необходимость динамического конструирования запроса UpdateSQL. Если вы не хотите это делать и используете ClientDataSet в своих приложениях, то вполне можно обойтись корректными настройками DataSetProvider (свойства ResolveDataSet и Options).

Вместо запросов в InsertSQL/UpdateSQL/DeleteSQL (и вообще их все необязательно заполнять, можно только часть, например, если удаление не допускается) можно указывать вызов хранимых процедур. Это позволяет "редактировать" запросы, которые представляют собой объединения нескольких таблиц, агрегаты (группировки) и т. п. случаи, когда нет прямого соответствия между записями в таблицах и записями, возвращаемыми запросом.

Внимание! В коде приложения для открытия и закрытия IBDataSet рекомендуется вызывать его методы Open/Close, а не управлять свойством Active. Во-первых, это почти одно и то же, а во-вторых, Open/Close лучше читаются и не допускают неоднозначностей (как например, IBDataSet.Active:=MyVar).

Описание методов ExecQuery, Prepare аналогично IBQuery, см. дальше.

Буферизация записей

Открытие запроса, возвращающего набор записей (Active:=True или вызов Open), приводит всего лишь к выполнению запроса, но не к выборке записей. Выборка записей начинается только тогда, когда происходит вызов методов Next, Last или FetchAll (Locate рассмотрен дальше, отдельно). Соответственно, свойство RecordCount показывает только число выбранных в буфер (с сервера) записей. IBCustomDataset и его наследники после Open выбирают только 1 запись в буфер, на чем выборка записей останавливается. Если DataSet подключен к DBGrid, то DBGrid, показывая записи, автоматически вызовет DataSet.Next столько раз, сколько поместилось записей на экран. При дальнейшем продвижении в DBGrid будут считываться остальные записи. По достижении конца выборки (последней записи) сервер сообщает клиентской части что записи кончились, после чего все выбранные записи находятся в буфере DataSet, и любое продвижение по записям

вверх или вниз (Next, Prev) приводит только к перемещению указателя внутри буфера DataSet (без обращения к серверу).

Если запрос не вернул ни одной записи, то оба свойства – BOF и EOF – будут равны True.

Если результат запроса обрабатывается по одной записи по мере их получения с сервера, и далее обработанные записи ни для чего не нужны, вместо кэширующих датасетов IBTable, IBDataSet и IBQuery следует использовать IBSQL – он не кэширует записи.

Обновление данных

У тех, кто уже знаком с механизмами работы технологии клиент-сервер, вопрос по обновлению данных, измененных другими приложениями, возникает только при неверно выбранном уровне изолированности (параметрах) транзакций (см. описание IBTransaction). У тех, кто только пришел с десктопных форматов БД (или вообще), и начинает работать с клиент-серверной СУБД впервые, возникает вопрос по обновлению данных другого рода – как это сделать вообще.

Если не вдаваться в подробности (также см. статью), то можно дополнить к предыдущему разделу (буферизация записей), что увидеть новые (committed) данные можно только выполнив запрос снова. Для этого, разумеется, нужно закрыть и открыть IBDataSet (или IBQuery/IBSQL). Переоткрытие (Close/Open) датасета приведет к уничтожению старого буфера записей, перевыполнению запроса, и считыванию записей в буфер. После чего опять до переоткрытия запроса изменения, выполненные другими приложениями (или в других транзакциях) опять будут "не видны".

Перебор записей

Для обработки всех записей, возвращаемых запросом, можно использовать цикл:

```
IBDataSet.Open;
```

```
while not IBDataSet.EOF do
```

```
begin
```

```
... // действия с текущей записью
```

```
IBDataSet.Next;
```

```
end
```

Аналогичный код будет работать и для IBTable, IBQuery.

Если перебор записей делается только с целью их однократной обработки, или экспорта, то вместо указанных компонент следует использовать IBSQL, который не кэширует записи. В противном случае при такой обработке запроса, возвращающего миллион записей, весь миллион записей окажется в памяти приложения.

Master-Detail

Странно, но такой вопрос часто возникает у разработчиков, которые начинают осваивать IBX. Если предполагать, что они перешли с BDE, то в IBX все делается точно так же, как в BDE master-detail на компонентах TQuery. Если опыта в подобных вещах нет, то вот пример:

1. кладем IBDatabase, IBTransaction, соединяем их
2. кладем 2 IBDataSet (IBQuery) и 2 TDataSource. Соединяем IBDataSet с IBDatabase, IBTransaction и DataSource
3. В базе есть 2 таблицы – клиенты и заказы. Клиенты – master, заказы – detail.
4. IBDataSet1 + DataSource1 называем MDS и MasterSrc соответственно. Прописываем у MDS в SelectSQL запрос

```
select * from clients
```

5. IBDataSet2 + DataSource2 называем DDS и DetailSrc соответственно. Прописываем у DDS в SelectSQL запрос

```
select * from orders where client_id = :cid.
```

Здесь имя параметра :cid должно совпадать с именем столбца первичного ключа в таблице clients. В данном примере этот столбец и называется CID.

6. у DDS указываем свойство DataSource = MasterSrc.
7. Активируем IBDatabase, IBTransaction, затем сначала MDS, затем DDS (можно еще проверить на форме, нажав правую кнопку и в меню CreationOrder, что компоненты создаются и активируются (открываются) в правильном порядке, и MDS+MasterSrc создаются раньше чем DDS+DetailSrc).
8. Подсоединяем 2 DBGrid, к MasterSrc и DetailSrc
9. Запускаем приложение. При перемещении по гриду master показываются соответствующие записи таблицы detail

В дополнение к master-detail хочется отметить еще один вопрос, который как таковой не относится к IBX. Иногда разработчики, не прочитав внимательно статью по генераторам, пытаются генерировать идентификатор мастера в триггере, в результате чего не могут ни откуда взять идентификатор мастера для вставки в таблицу detail. Само-собой, идентификатор надо сначала получить (например, установкой свойства GeneratorField у IBDataSet или IBQuery), использовать для вставки запись в master-таблицу, а затем в таблице detail этот идентификатор "возьмется" автоматически. Или он уже и так будет известен даже при ручной вставке записи в таблицу detail.

Кроме того, при работе с `CachedUpdates` (если вам вообще нужен режим `CachedUpdates`) необходимо тщательно следить за порядком отправки записей на сервер (как в пункте 7 обращено внимание на порядок создания компонент) – вначале должна идти запись `master`, и только затем – `detail`. Для этого нужно вызывать методы `ApplyUpdates` соответствующих компонент в правильном порядке (а не вызывать `IBDatabase.ApplyUpdates`, т. к. в этом случае правильный порядок может быть не соблюден).

Locate (поиск)

В выбранных `IBDataSet` (`IBQuery`) записях можно производить поиск вызывая метод `Locate`. Поскольку запрос, выбирающий записи, указан явно в `SelectSQL`, компонент не может сделать ничего другого как по очереди перебирать выбранные записи от начала до конца (от `First` до `EOF`, путем вызова `Next`), и производить поиск в буфере. Если `IBDataSet` был только что открыт, и `FetchAll` не вызывался, то вызов `Locate` приведет к выборке на клиента (в локальный буфер) всех записей, возвращаемых запросом.

Сортировка

Сортировка результата, а также "вставка новых записей в правильное место" у компонент `IBDataSet` и `IBQuery` невозможна. Они не имеют понятия, каким образом данные отсортированы запросом, выполнявшимся на сервере, и отсортированы ли они вообще. Результат выполнения запроса считывается в буфер `IBDataSet/IBQuery`, и выводится далее только в том порядке, в котором записи были считаны. Если вам нужна сортировка данных "по столбцам", следует использовать `TClientDataSet`, или гриды, которые могут самостоятельно сортировать данные в буфере (например, `VirtualTreeView`).

Работа с BLOB

Ничего сложного. В IBX это делается следующим образом (то же самое – и в `FIBPlus`. См. статью <http://www.devrace.com/ru/fibplus/articles/2261.php>)

```
procedure TForm1.Button1Click(Sender: TObject);  
begin
```

```
IBDataSet1.FieldByName('BLB') as TBlobField).SaveToFile('c:\blob.bin');
```

```
...
```

```
end;
```

Практически то же самое – при записи, причем делать это можно двумя вариантами:

1. передача blob через параметр. Имеется в виду запрос вида

```
insert into table (field1, field2, blbfield) values (:param1, :param2, :blb); :
```

```
IBQuery1.ParamByName('blb').asBlob:=blobvar;
```

asBlob принимает значение в виде простой строки (string). Действительно, строки в последних версиях Delphi (как минимум с Delphi 3) могут содержать любую, в т. ч. двоичную информацию. Поэтому значение параметру blob можно присваивать и как asString.

2. изменение столбца "редактируемого" DataSet – запись blob из файла в столбец:

```
IBDataSet1.Edit;  
(IBDataSet1.FieldByName('BLB') as TBlobField).LoadFromFile('c:\blob.bin');  
IBDataSet1.Post;
```

В предыдущих версиях IBX (до 4.42 включительно) можно было работать с blob иначе. Например, запись в файл

```
procedure TForm1.Button1Click(Sender: TObject);  
var B: TIBBlobStream;  
begin
```

```
B:=IBDataSet1.CreateBlobStream(IBDataSet1.FieldByName('BLB') as TBlobField,  
bmRead);  
B.SaveToFile('c:\blob.bin');  
B.Free;  
  
end;
```

и чтение из файла

```
IBDataSet1.Edit;  
B:=IBDataSet1.CreateBlobStream(IBDataSet1.FieldByName('BLB') as TBlobField,  
bmWrite);  
B.LoadFromFile('c:\blob.bin');  
B.Free;  
IBDataSet1.Post;
```

Обратите внимание, что метод B.Free вызывается перед вызовом IBDataSet1.Post (а не наоборот, как это приведено в некоторых старых примерах в интернете).

JPEG в Blob

Какой формат хранить в Blob – абсолютно неважно. Компоненты IBX и Interbase/Firebird никак не реагируют на содержимое Blob – они его не анализируют, не отображают, не конвертируют и т. п. Если у вас есть проблемы с отображением jpeg, хранимого в blob, обратитесь к поисковым серверам в

интернете – вы за 1 минуту сможете найти несколько вариантов DBImage, который корректно работает с Jpeg и другими форматами. Еще раз подчеркиваю, что любые проблемы с форматом содержимого blob не относятся к IBX, FIBPlus, любым другим драйверам, и к серверам InterBase, Firebird или Yaffil.

Blob и DBGrid

Стандартный DBGrid не показывает в строке содержимое blob. Различные сторонние варианты DBGrid могут отображать – наличие-отсутствие данных в blob, значение blob в виде hint, в виде строки, в виде картинки и т.п. Практически все эти способы являются малоэффективными с точки зрения визуализации данных, и на самом деле производят серьезную нагрузку на сеть из-за перекачивания всех столбцов blob с сервера на клиента.

Blob-данные в записях не хранятся. То есть, в записи (столбце) на самом деле находится ссылка на blob. И клиентская часть извлекает с сервера сначала записи, и только потом, если клиентские компоненты это делают, при помощи специальных вызовов по идентификатору blob он выбирается с сервера. То есть, даже если в записи есть blob-столбцы, то их выборка – дело совершенно необязательное.

Если перед вами стоит задача отображения информации вроде записей отдела кадров, где к записи о работнике могут быть прицеплены следующие (и иногда весьма объемные) данные – автобиография, резюме, фотография 3х4, фотография 9х12, договор, и т. п.

Допустим, в IBDataSet или IBQuery вы написали запрос `select * from personinfo`. Стандартный DBGrid, как уже говорилось, никак не умеет выводить содержимое столбцов blob. Поэтому blob-ы при выполнении данного запроса не будут считываться с сервера. Если же DBGrid показывает blob-ы в том или ином виде, то соответственно, все блобы из вместилища в DBGrid записей будут считаны на клиента. Возможно, что более эффективным способом было бы вместо `select *` написать `select` с перечислением только тех столбцов, которые действительно нужны, а затем, при выборе нужной записи, показывать содержимое столбцов blob либо автоматически, по таймеру, либо по нажатию специальной кнопки (на этой же форме или в отдельной, как угодно), отдельным запросом

```
select blob1, blob2, blob3 from table where pk_field = :param
```

В параметр, разумеется, записывать значение столбца первичного ключа, полученного из той записи, на которой находится пользователь.

Примечание. "По таймеру, ..." – имеется в виду следующий механизм: на форму с DataSet/DataSource кладется TTimer, у которого выставляется интервал срабатывания в 800 (0.8 секунды, зависит от квалификации пользователя программы). TTimer постоянно сбрасывается и взводится снова в событии DataSource.OnDataChange (для режима просмотра записей, dsBrowse). По истечению таймера (TTimer.OnTimer), если в DataSet есть записи, у текущей записи берется значение столбца первичного ключа, выполняется вышеупомянутый запрос, и компоненты Memo, Picture и т. п. заполняются данными, считанными этим дополнительным запросом.

IBTable

TIBTable = class(TIBCustomDataSet)

IBTable – в общем случае, заменитель компонента TTable для облегчения перехода с BDE (как и IBQuery). Для работы требует указания имени таблицы, после чего запрос на считывание данных формируется автоматически.

Важные свойства:

- **TableName** – имя таблицы
- **TableTypes** – типы таблиц, отображаемые в выпадающем списке TableName – ttSystem включает выборку системных таблиц (rdb\$, tmp\$), а ttView – включает выборку VIEW.
- **IndexName** – имя индекса для сортировки. Так же, как и в BDE, имя индекса используется косвенно, для получения столбцов индекса, и добавления фразы ORDER BY INDEXFIELD1, INDEXFIELD2 к автоматически конструируемому запросу. Поэтому для сортировки по любому столбцу, даже непроиндексированному, следует использовать свойство IndexFieldNames.

Явно сортировку записей в обратном порядке (DESC) указать невозможно.

Filter – условие фильтрации. Текст свойства Filter добавляется как условие where к автоматически конструируемому запросу.

Поскольку свойство ReadOnly по умолчанию False, IBTable сразу готов к редактированию данных – раз запрос на чтение формируется из одной таблицы, то запросы insert/update/delete могут быть легко сформированы автоматически. Сортировка записей по умолчанию (order by) производится по столбцам первичного ключа (primary key).

Locate

Для IBTable Locate производит поиск несколько иначе, чем например для IBQuery.

Условия поиска раскладываются в соответствующую конструкцию where, после чего выполняется запрос. Если такие записи найдены, то поиск производится вызовом `IBCustomDataSet.Locate` (аналогично `IBDataSet`, `IBQuery`). Если записи не найдены, поиск в буферизированных записях не производится.

Почему компонент `IBTable` не рекомендуют?

Действительно, на многих форумах использование `IBTable` не рекомендуют и считают "ламерским". Дело в том, что как уже было сказано выше, `IBTable` формирует SQL-запросы к БД самостоятельно. А это при работе с SQL-серверами считается некорректным. Т. е. запросы необходимо контролировать, для того чтобы знать, что именно делается для выборки данных. Более того, часто `IBTable` используют для выборки из таблиц с большим числом записей, что также отрицательно влияет на производительность сервера и приложения.

В основном, конечно, негативное отношение к `IBTable` базируется на аналогичном отношении к его "родителю", компоненту `TTable`, который мог приводить к серьезным проблемам с производительностью приложений.

Итак, `TTable` предназначен для имитации "навигационного" доступа, к которому привыкли пользователи десктопных СУБД, и плох для SQL-сервера. `IBTable` наследует его характеристики, и предназначен для облегчения переноса приложений BDE на компоненты IBX. Тем не менее, иногда удобно использовать `IBTable` для осуществления доступа к справочникам.

Помните, что изначально в компонентах `FreeIBDatabase` никакого `IBTable` не было. Был только один компонент, который позволял работать с набором данных – `IBDataSet`. И уже от него произошли и `IBQuery`, и `IBTable`.

IBQuery

```
TIBQuery = class(TIBCustomDataSet)
```

`IBQuery` – компонент для выполнения запросов. Если не требуется выполнять "редактирование" записей запроса, то `IBQuery` можно использовать вместо `IBDataSet`. Или, как замену `IBDataSet` можно использовать комбинацию `IBQuery + IBUpdateSQL`.

Дополнительно к свойствам `IBDataSet` (кроме отсутствующих `Refresh/Insert/Update/DeleteSQL`) имеет свойство **Constraints** (отсутствие данного свойства у `IBDataSet` скорее связано с какими то недоделками, т. к. компоненты `IBTable`, `IBQuery` и `IBDataSet` – унаследованы от `IBCustomDataSet`).

Для запросов, возвращающих набор записей, можно установить свойство Active в True или вызвать метод Open. Для остальных запросов, таких как insert/update/delete, execute или операторов DDL (create table, alter, drop...) нужно вызывать метод ExecQuery, т. к. эти запросы не возвращают записи, а возвращают только результат выполнения оператора SQL.

Помните, что IBQuery при выполнении запросов SELECT буферизирует записи точно так же, как и IBDataSet. Если необходимо просто перебрать записи, или экспортировать их в файл или другую СУБД – используйте небуферизирующий компонент IBSQL.

Внимание! Для выполнения процедур, возвращающих данные (execute procedure) следует использовать IBSQL или IBStoredProc, т.к. вызов IBQuery.ExecSQL не заполняет Fields данными, полученными от процедуры.

Параметризированные запросы

Компоненты IBDataSet, IBQuery и IBSQL могут выполнять как статический, так и динамический SQL. Динамический SQL отличается от статического наличием параметров. Пример статического SQL:

```
select * from table
```

```
where field > 5
```

Если вместо цифры 5 предполагается использовать значение, полученное как ввод данных пользователем, то следует использовать параметризированный запрос (ParamCheck:=True):

```
select * from table
```

```
where field > :param
```

Двоеточие или символ '?' означают, что запрос предполагает задание параметра. Делается это следующим образом:

```
IBQuery.SQL.Clear; // очистить текст sql
```

```
IBQuery.SQL.Add('select * from table where field > :param'); // задать текст запроса
```

```
IBQuery.Prepare; // отправить запрос на сервер, проверить его корректность и т. п.
```

```
IBQuery.ParamByName('param').asInteger:=5; // задать значение параметра
```

```
IBQuery.Open; // или IBQuery.ExecSQL
```

Для статических запросов вызов Prepare необязателен – компонент сам его выполнит автоматически, если Prepare не был вызван.

Примечание. После Prepare можно обратиться к свойству Plan, т.к. именно после Prepare сервер сообщает план выполнения запроса.

Prepare очень удобен при повторяющемся выполнении одного и того же запроса, с разными значениями параметра. При этом Prepare вызывается один раз, а установка параметров и вызов ExecQuery производится столько раз, сколько нужно. Чаще всего такой способ используется для запросов Insert и Update.

Внимание! Имена параметров поддерживаются только клиентской библиотекой. То есть, сервер понимает параметры только в виде символа '?'. Если компоненты умеют обрабатывать именованные параметры, то они перед отправкой запроса на сервер "вырезают" их, отправляя туда запрос вида `select * from table where field > ?`

Некоторые библиотеки компонент могут не поддерживать или некорректно обрабатывать запросы, в которых используется два или более параметров с одинаковым именем.

Фильтрация

Возможно, в BDE или ином используемом вами до IBX наборе компонент доступа к БД вы активно применяли фильтрацию записей. В общем фильтрация в IBX работает практически так же, как и в BDE.

IBTable

Для компонента IBTable в свойстве Filter должны быть описаны условия фильтрации так, как будто вы добавляете условие where к обычному запросу SQL. IBTable сконструирует запрос, используя заданное в TableName имя таблицы, добавит как where условие фильтрации из Filter, и добавит конструкцию ORDER BY если указана "сортировка" результирующего набора при помощи свойств IndexFieldNames или IndexName. Поэтому в Filter должен быть указан текст с учетом всех особенностей синтаксиса InterBase или Firebird. Например,
LastName like 'A%'

В итоге запрос, отсылаемый на сервер, будет иметь вид
select id, LastName, FirstName...
from TABLE
where LastName like 'A%'

IBDataSet, IBQuery

Компоненты IBDataSet, IBQuery выполняют запросы, четко заданные в свойствах SelectSQL или SQL. Поэтому дополнительная фильтрация возможна только методом OnFilterRecord.

OnFilterRecord

Компоненты IBTable, IBDataSet и IBQuery позволяют использовать это событие для

фильтрации записей. В отличие от `IBTable`, где в свойстве `Filter` можно указать дополнительное условие отбора для запроса, выполняемого на сервере, `OnFilterRecord` определяет "видимость" записи, уже принятой `DataSet`-ом с сервера. Метод имеет два параметра – `DataSet`, то есть что за компонент запросил фильтрацию (для возможности получения данных из "текущей" записи), и `Accept`, который определяет, показывать данную запись подключенному компоненту `DataSource` или нет. Для того, чтобы имитировать фильтрацию, показанную выше на примере для `IBTable`, в `OnFilterRecord` нужно написать примерно следующее: `Accept:=(DataSet.FieldByName('LastName').asString >= 'A') and (DataSet.FieldByName('LastName').asString < 'B');`

Помните, что фильтрация производится в локальном буфере уже принятых с сервера записей. То есть, если в программе предполагается небольшое число вариантов фильтрации, то возможно, имеет смысл уменьшить нагрузку на клиентскую часть и при изменении фильтра конструировать запрос для `SelectSQL` с нужным условием `where` (точно так же как это делает `IBTable`, только строку `where` вы будете добавлять к тексту запроса самостоятельно).

IBUpdateSQL

`IBUpdateSQL` – компонент для "оживления" `IBQuery`. То есть, комбинация `IBQuery` + `IBUpdateSQL` представляет собой аналог `IBDataSet`. Для работы такой связки не требуется включение режима `CachedUpdates`, как это требовалось в `BDE`.

IBUpdateSQLW

В поставке компонента `IBUpdateSQLW` нет. Зато он есть [здесь](#). Он предназначен для выполнения операторов `select/refresh` и `insert/update/delete` в разных транзакциях. Зачем это нужно, см. дальше информацию по транзакциям в приложении.

IBSQL

`TIBSQL = class(TComponent)`

`IBSQL` – компонент, являющийся минимальной надстройкой над `InterBase API`. В отличие от `IBTable`, `IBQuery` и `IBDataSet` он не буферизирует выбираемые записи – для доступа к текущей записи существует только свойство `Current` (`IBSQL` не является наследником `DataSet`). Работа с параметрами и столбцами несколько отличается от обычной, т. к. они являются `TIBXSQLVAR`, то есть прямыми ссылками на структуры `InterBase API`.

У IBSQL есть свойство GoToFirstRecordOnExecute – если True (по умолчанию), то в случае select компонент запросит у сервера первую запись и заполнит ее данными Current. Если False – Current будет пустым, и для считывания данных с сервера необходимо выполнить IBSQL.Next.

Предназначен для выполнения операторов Insert/update/delete, execute procedure, или select для небуферизированного перебора записей (например при импорте/экспорте данных или при массовой обработке записей на клиенте).

Пример:

```
IBSQL1.SQL.Clear;
IBSQL1.SQL.Add('select * from table');
IBSQL1.GoToFirstRecordOnExecute:=True;
IBSQL1.ExecQuery;
while not IBSQL1.Eof do

begin

// получение данных столбца в переменную
...:=IBSQL1.Current.By Name('fieldname').as...;
IBSQL1.Next;

end;
```

IBStoredProc

```
TIBStoredProc = class(TIBCustomDataSet)
```

IBStoredProc – компонент для выполнения хранимых процедур. Собственно, выполнять процедуры можно вызывая EXECUTE PROCEDURE в компонентах IBQuery и IBSQL. Унаследован от IBCustomDataSet, но как таковым DataSet-ом не является. Поэтому для выборки из селективных процедур (select * from proc) следует использовать обычные IBDataSet, IBQuery, IBSQL (IBStoredProc принципиально не умеет работать как DataSet по простой причине – выполнение процедур в нем предусмотрено только как execute procedure). В общем случае не рекомендуется к использованию, т. к. имеет застарелую проблему с обработкой ошибок. При самостоятельном исправлении данной ошибки в коде (IBStoredProc.pas) может быть использован, однако при установке обновлений IBX нужно проверить новую версию IBStoredProc на исправления (что маловероятно, т.к. ошибка остается уже много лет).

Примеры вызова обычных и селективных процедур через IBDataSet, IBQuery и IBSQL даны в статье.

Внимание! Помните, что указанная выше проблема с обработкой ошибок относится к вызовам `execute procedure` во всех компонентах IBX – `IBStoredProc`, `IBQuery`, `IBDataSet`, `IBSQL`, ...

EIBError

EIBError – базовый класс ошибок IBX. Имеет свойства (кроме унаследованных от `EDatabaseError`):

- **SQLCode** – номер ошибки (статус вектора) при выполнении операторов SQL
- **IBErrorCode** – целое число, идентифицирующее ошибку. Описание всех ошибок сервера находится в `LangRef.pdf` (а также перечислено в `IBErrorCodes.pas`)

В свою очередь, на основе EIBError в IBX существует несколько дополнительных классов:

- `EIBInterBaseError` – общая ошибка сервера
- `EIBInterBaseRoleError` – ошибки при использовании `ROLE`
- `EIBClientError` – ошибки, специфичные для клиентской части (`gds32.dll`, `fbclient.dll`). В основном для указания на несовместимости конкретной клиентской части.
- `EIBPlanError` – ошибка получения плана запроса

К сожалению, затея с разделением EIBError на классы хоть и хорошая, но в коде реализована плохо. Например, `EIBPlanError` и `EIBInterBaseRoleError` вызываются в коде только по одному разу. Поэтому необходимо проверять EIBError, а наследованные от него классы можно игнорировать, или использовать только `EIBInterBaseError` и `EIBClientError` в дополнение к EIBError.

Примеры обработки ошибок при работе с транзакциями [приведены дальше](#).

IBDatabaseInfo

IBDatabaseInfo – компонент для получения информации о параметрах БД и статистике выполнения запросов. Подсоединяется к [IBDatabase](#), информацию возвращает только посредством `runtime`-свойств. Наиболее часто используемые свойства:

- **UserNames** – список пользователей, подключенных к серверу архитектуры `SuperServer`. Для `Classic` возвращает только текущего пользователя.
- **Reads, Writes, Fetches** – информация о числе чтений, записи и обращений к страницам БД. Аналог информации, выводимой `IBExpert` и др. инструментами после выполнения запроса.

- **PageSize, ODSxxx, NumBuffers, SweepInterval, ...** – информация о свойствах БД.
- **ReadSeqCount, UpdateCount, InsertCount, ...** – счетчики соответствующих операций, выводятся для всех таблиц (поштучно). В предыдущих версиях IBX представляли собой integer, что неверно (выводился мусор вместо правильной информации). Аналог информации, выводимой в закладке Performance Info IBEExpert.

IBSQLMonitor

IBSQLMonitor – компонент для мониторинга действий клиентского приложения (вызовов IB API). Для мониторинга необходимо в событии OnSQL сохранять EventText и EventTime либо в файле, либо в компоненте TМемо на какой-либо форме (например, Мемо.Lines.Add(EventText)). Возможен мониторинг (свойство TraceFlags):

- tfQPrepare – вызов Prepare запроса
- tfQExecute – вызов Execute запроса. Выдается текст запроса, хотя на самом деле в этот момент он на сервер не передается (был уже передан при Prepare)
- tfQFetch – выборка записей
- tfError – ошибки, возникающие на сервере
- tfStmt – операции с запросами (аллокирование, закрытие, и т. п.)
- tfConnect – подключение к базе данных или отключение
- tfTransact – операции с транзакциями – Start, Commit, Rollback, CommitRetaining, RollbackRetaining
- tfBlob – операции с blob (чтение, запись)
- tfService – вызовы Services API
- tfMisc – остальное

Для активации монитора нужно у соответствующего компонента IBDatabase также выставить опцию TraceFlags.

В некоторых (старых) версиях IBX были проблемы с мониторингом (зависало приложение), если компонент IBSQLMonitor использовался в приложениях, работающих как сервис.

Пример использования есть в поставке, и называется SQLMonitor.

IBEvents

IBEvents – компонент для регистрации и приема событий, отправляемых сервером по оператору POST_EVENT. См. пример IBDemo в поставке Delphi и C++Builder. Помните, что события синхронные, то есть отправляются клиенту только по commit транзакции, в которой эти события произошли (до commit или по rollback никакие

события не отправляются).

IBExtract

IBExtract – компонент для извлечения метаданных БД (таблиц, процедур, триггеров...) в виде скрипта.

Основные свойства:

- IncludeSetTerm – включать команду set term в скрипт, или нет. Необходимо (true) для последующего выполнения скрипта в isql, IBExpert и др.
- ShowSystem – выводить в скрипт ddl системных таблиц (rdb\$...)
- DatabaseInfo (runtime) – локальный экземпляр IBDatabaseInfo
- Items – результирующий скрипт с метаданными
- Database – связь с компонентом IBDatabase
- Transaction – связь с компонентом IBTransaction

Примеры использования:

Извлечь полный скрипт:

```
IBExtract1.ExtractObject(eoDatabase);  
IBExtract1.Items.SaveToFile('d:\a.ddl');
```

Извлечь только скрипт таблиц:

```
IBExtract1.ExtractObject(eoTable);  
IBExtract1.Items.SaveToFile('d:\a.ddl');
```

Извлечь данные для таблицы Customer:

```
IBExtract1.ExtractObject(eoData, 'CUSTOMER');  
IBExtract1.Items.SaveToFile('d:\a.ddl');
```

Извлечь скрипт таблицы EMPLOYEE:

```
IBExtract1.ExtractObject(eoTable, 'EMPLOYEE');  
IBExtract1.Items.SaveToFile('d:\a.ddl');
```

Извлечь скрипт таблицы EMPLOYEE плюс все ее индексы, триггеры, домены, гранты, check и данные:

```
IBExtract1.ExtractObject(eoTable, 'EMPLOYEE',  
[etDomain, etTable, etTrigger, etForeign, etIndex, etData, etGrant, etCheck]);  
IBExtract1.Items.SaveToFile('d:\a.ddl');
```

Извлечь скрипт процедуры SHIP_ORDER:

```
IBExtract1.ExtractObject(eoProcedure, 'SHIP_ORDER');  
IBExtract1.Items.SaveToFile('d:\a.ddl');
```

Извлечь скрипт (create и alter) процедуры SHIP_ORDER:

```
IBExtract1.ExtractObject(eoProcedure, 'SHIP_ORDER');  
IBExtract1.Items.SaveToFile('d:\a.ddl');
```

Извлечь скрипт alter procedure для SHIP_ORDER:

```
IBExtract1.ExtractObject(eoProcedure, 'SHIP_ORDER', etAlterProc);  
IBExtract1.Items.SaveToFile('d:\a.ddl');
```

IBConnectionBroker

IBConnectionBroker – компонент для организации пула коннектов. Пример использования – [IBConnectionBroker](#). Для нормальной работы (параллельной работы с коннектами в threads) нельзя указывать локальный коннект (без имени сервера. Это допустимо только если используется Firebird Embedded версии 1.5). В случае работы приложения с IBConnectionBroker на сервере укажите строку коннекта у [IBDatabase](#) как localhost:c:\dir\data.gdb.

IBScript

IBScript – компонент для выполнения скриптов (набора команд в текстовом файле).

Основные свойства:

- AutoDDL – автоматически вызывать Commit после выполнения DDL-операторов
- Dataset – компонент для выполнения sql-операторов скрипта. Рекомендуется [IBQuery](#).
- Database – база данных, над которой нужно выполнить скрипт ([IBDatabase](#))
- Transaction – транзакция, в которой будет выполняться скрипт ([IBTransaction](#))
- Terminator – разделитель SQL-операторов в скрипте
- Script – текст скрипта (TStrings, соответственно можно вызывать LoadFromFile и т. п.)
- Statistics – статистика выполнения скрипта (подробнее см. IBScript.pas)
- Paused (runtime, read/write) – индикатор состояния выполнения скрипта
- Validating (runtime, readonly) – индикатор выполнения проверки скрипта (только prepare)
- CurrentTokens (runtime) – набор токенов выполняемого SQL-оператора

События:

- OnExecuteError – при ошибке выполнения конкретного оператора позволяет вывести этот оператор, номер строки в скрипте и ошибку, а также указать, прервать выполнение скрипта или игнорировать ошибку и выполнять скрипт дальше.
- OnParse – при успешном выполнении очередного оператора скрипта позволяет узнать тип оператора (TIBParseKind) и его текст.
- OnParseError – при ошибке парсинга оператора позволяет вывести оператор, ошибку и номер строки в скрипте.
- OnParamCheck – если у конкретного оператора есть параметры, и это не DDL, то позволяет задать параметры оператора SQL перед его выполнением (Params и ParamByName).

Методы:

- ValidateScript – выполнять только prepare всех операторов скрипта, чтобы убедиться, что они могут быть выполнены (синтаксически корректны).
- ExecuteScript – выполнить скрипт.

IBSQLParser

IBSQLParser – компонент, которым пользуется IBScript для парсинга скрипта.

IBDatabaseINI

IBDatabaseINI – компонент для записи и считывания настроек IBDatabase в ini-файле.

Примеры использования:

Сохранение информации:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin
```

```
IBDatabaseINI1.ReadFromDatabase;  
IBDatabaseINI1.SaveToINI;
```

```
end;
```

Считывание информации:

```
procedure TForm1.Button2Click(Sender: TObject);  
begin
```

```
IBDatabaseINI1.ReadFromIni;
```

```
IBDatabaseINI1.WriteToDatabase(IBDatabase1);  
  
end;
```

Информация хранится в файле, указанном в свойстве FileName, в виде

[Database Settings]

database=localhost:D:\Firebird\examples\employee.fdb

user_name=SYSDBA

password=masterke

sql_role=

lc_ctype=win1251

Классы импорта/экспорта в IBSQL.PAS

Перенесены и чуть доработаны еще из FreeIBComponents.

Базовым классом является TIBBatch. От него унаследованы TIBBatchInput и TIBBatchOutput, которые также являются промежуточными. Конечные (готовые для использования) классы:

TIBOutputDelimitedFile, TIBInputDelimitedFile – импорт-экспорт в файлы текстового формата, каждая запись на отдельной строке, столбцы нефиксированного формата, возможность указать разделитель.

Пример импорта:

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
  
    Filename : String;  
    InpFile : TIBInputDelimitedFile;  
  
begin  
  
    TW.StartTransaction;  
    ImpFile:=TIBInputDelimitedFile.Create;  
    ImpFile.Filename:='c:\data.csv'; ImpFile.ColDelimiter:=';'; // разделитель столбцов  
    ImpFile.SkipTitles; // пропускаем заголовок с именами столбцов, если есть  
    ImpSQL.Transaction:=TW; // связываем компонент с транзакцией, если это не  
    сделано в дизайн-тайме  
    ImpSQL.Clear; // это класс IBSQL. Можно использовать и IBQuery  
    ImpSQL.SQL.Add('insert into table (field1, field2, field3) ');  
    ImpSQL.SQL.Add('values (:field1, :field2, :field3) ');
```

```
ImpSQL.BatchInput(ImpFile);  
ImpFile.Free;  
TW.Commit;  
  
end;
```

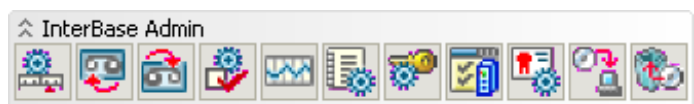
Для экспорта используется соответственно запрос select, TIBOutputDelimitedFile, и метод IBSQL.BatchOutput(). Несмотря на то, что у TIBQuery/TIBDataSet тоже есть метод BatchOutput, эти компоненты не рекомендуется использовать для экспорта. Дело в том, что IBQuery/IBDataSet – кэширующие DataSet-ы. То есть, по мере считывания и экспорта записей поштучно они будут накапливаться в буфере записей, и будет расти используемая клиентским приложением, и падать скорость экспорта. IBSQL наоборот, не имеет буфера (есть буфер только на 1 запись), и поэтому память при экспорте потребляться не будет, а скорость считывания записей будет зависеть только от производительности сервера и сети.

TIBOutputRawFile, TIBInputRawFile – импорт-экспорт в файлы фиксированного формата, без разделителей, аналог external table (собственно, можно экспортировать данные в такой файл и использовать его как external table). К этим компонентам относится все то же самое, что было сказано выше в отношении TIBxxxDelimitedFile.

TIBOutputXML – класс для экспорта в XML. Скорее, это не класс, а record, или набор параметров для экспорта в XML, поскольку никаких методов или функций у TIBOutputXML (кроме наследуемых от TObject) нет.

Для экспорта в XML не требуется каких то специфических действий – достаточно создать экземпляр TIBOutputXML, заполнить его свойства, и вызвать TIBSQL.OutputXML();

Компоненты Services API (InterBase Admin)



Этот набор компонент находится на отдельной закладке, и показывается в палитре компонент Delphi только в том случае, если на компьютере в path присутствует gds32.dll от InterBase 6.0 и выше или совместимая (для Firebird обязательно указание установки совместимой gds32.dll в инсталляторе, или ручная установка gds32.dll из fbclient.dll при помощи утилиты instclient.exe). Компоненты используют Services API, введенный в InterBase 6.0 для backup/restore, проверки БД и других операций. Пример использования этих компонент есть в поставке и называется Admin (если вы не нашли этот пример в Demos/DB/IBX/, то можете скачать [отсюда](#)).



Services API как таковое должно поддерживаться сервером, и поддерживается во всех версиях архитектуры SuperServer, начиная с InterBase 6. Для Classic полноценно Services API поддерживается либо в Yaffil, либо в Firebird начиная с версии 1.5.2.

На картинке изображена форма примера Admin в DesignTime.

Примечание. Services API в данном случае – это набор сервисных функций СУБД Firebird или InterBase. И к сервисам операционной системы он не имеет ни малейшего отношения.

У всех компонент Services API подключение к серверу и БД производится двумя свойствами – ServerName и DatabaseName. **В отличие от TIBDatabase в свойстве DatabaseName должен быть указан только локальный путь к БД (без имени сервера), а в ServerName – только имя сервера.** Например,
IBS.DatabaseName:='c:\dir\data.gdb';
IBS.ServerName:='server';

Если в DatabaseName указать и имя сервера ('server:c:\dir\data.gdb'), компонент будет выдавать ошибку 'Error reading data from the connection'.

Все компоненты Services API предназначены для выполнения сервером определенных команд. Сами компоненты (за исключением TIBInstall, TIBUninstall) никаких действий (кроме передачи команд серверу и получения результата) не производят.

У всех компонент (за исключением TIBInstall, TIBUninstall) имя пользователя и пароль задаются в свойстве Params. Например,

```
IBBackupService1.Params.Add('user_name=SYSDBA');  
IBBackupService1.Params.Add('password=masterkey');
```

TIBConfigService – позволяет получать параметры сервера из файла конфигурации ibconfig.

TIBBackupService – передает серверу команду на выполнение резервного копирования базы данных, аналог `gbak -b` с опцией `-se`. Файл бэкапа создается сервером на том же компьютере, где находится сервер и база данных. В случае необходимости создать резервную копию на другом компьютере, необходимо, чтобы сервер стартовал не под LocalSystem, а под специально созданным пользователем (т. к. LocalSystem не имеет и не может получить никаких прав на доступ к сетевым ресурсам), и этому пользователю были даны права на доступ к удаленному (сетевому) каталогу, куда требуется сохранять резервную копию.

TIBRestoreService – передает серверу команду на восстановление базы данных из резервной копии, аналог `gbak -c` с опцией `-se`. Изложенное выше для IBBackupService справедливо и для этого компонента.

TIBValidationService – аналог `gfix`, осуществляет проверку БД и/или ремонт повреждений в БД, если таковые присутствуют.

TIBStatisticalService – аналог `gstat`, получает статистику из базы данных. См. [IBAnalyst](#).

TIBLogService – позволяет получить содержимое interbase.log с сервера.

TIBSecurityService – управляет пользователями в `isc4.gdb`, `admin.ib` или `security.fdb` (посредством сервера, разумеется). Позволяет добавлять, удалять, модифицировать учетные записи пользователей, а также выводить список зарегистрированных пользователей.

TIBServerProperties – позволяет получить информацию о сервере, установленных лицензиях. конфигурационные параметры.

TIBLicensingService – работает только с платными версиями InterBase. Позволяет управлять пользовательскими и процессорными лицензиями на сервере (также серверными для 7.0. В 7.1 и 7.5 серверные и unlimited users лицензии регистрируются только через инсталлятор). IBConsole для управления лицензиями использует именно этот компонент.

TIBInstall, TIBUninstall – компоненты, обеспечивающие установку, регистрацию и деинсталляцию InterBase в системе.

Использование и управление IBTransaction в приложениях

Существование IBTransaction дает возможность не только управлять параметрами транзакций целиком и полностью, но и полноценно управлять стартом и завершением транзакций. Возможно, если вы раньше работали с BDE, где практически никакого управления транзакциями нет (одна транзакция на коннект), и ваша система имела мало пользователей (до 10-20-ти), то вы не испытывали никаких проблем. Действительно, BDE самостоятельно стартует и завершает транзакции в нужный момент. Однако, вся проблема в том, что получить эффективную многопользовательскую систему без явного управления транзакциями можно только в том случае, когда промежуточное звено (компоненты, BDE и т. п.) гарантирует, что длительность транзакций будет минимальна. Если же управлять транзакциями явно, то можно создать еще более эффективную систему. В противном случае "автоматизированность" управления транзакциями приведет к тому, что приложения будут чаще попадать на блокировки записей, сервер будет страдать от большого числа одновременно активных транзакций, или в базе будет накапливаться мусор из-за того, что какая-то транзакция активна с самого утра рабочего дня.

Весь ужас в том, что при отсутствии контроля над транзакциями вы не можете понять, что происходит. Производительность системы может деградировать с каждым часом, и сделать практически ничего нельзя (кроме рестарта сервера и обретения спокойствия еще на несколько часов).

Конечно, InterBase 7.5 и выше или Firebird 2.1 и выше дадут вам возможность выяснить (при помощи временных системных таблиц tmp\$ или mon\$ соответственно), что же за транзакция активна уже 3 часа, и сколько их вообще таких существует. И можно эти транзакции принудительно завершить (или отловить пользователей, которые открывают ненужную форму, или не закрывают приложения уходя с работы). Но выяснить можно, а вот исправить – нет. То есть, необходимость переделки пользовательских приложений зависнет над вашей головой как дамоклов меч.

Поскольку, как уже было сказано, IBX (как и FIBPlus) позволяет полноценно управлять транзакциями, следует соблюдать несколько правил:

1. Никогда не полагайтесь на defaultTransaction, неявный старт транзакций или их автоматическое завершение. У вас должна быть всегда возможность выяснить, в каком месте кода и с какими параметрами стартует конкретная транзакция, и когда завершается. Это увеличивает количество строк кода? Да, несомненно, но не намного, однако работа ваших приложений улучшится в геометрической прогрессии. При этом в коде будет четко видно, какие операции выполняются в транзакции (если, конечно, не вызывать Commit/Rollback в другом модуле относительно StartTransaction).

2. Не рекомендуется стартовать или завершать транзакции обращением к свойству IBTransaction.Active. В описании IBTransaction об этом уже было сказано, но стоит повторить еще раз – привычка вызывать в коде Active так же, как

вы это делаете в дизайн-тайме (кликая на свойство active в Object Inspector) может сослужить плохую службу – если транзакция в IBTransaction уже активна, то при вызове IBTransaction.Active:=False; транзакция безусловно завершится по Rollback (это можно легко увидеть в коде IBDatabase.pas, метод TIBTransaction.SetActive). Поэтому, а также для улучшения читаемости кода, свойство Active рекомендуется использовать только для проверки состояния компонента (для этого есть свойство inTransaction, кстати). А старт и завершение транзакции лучше всегда писать явно, используя методы StartTransaction, Commit или Rollback.

Примечание. В FIBPlus свойство TimeoutAction (аналогичное DefaultAction в IBX) равно taRollback. Это нужно учитывать, особенно если вы планируете заменить IBX на FIBPlus. Кстати, если при закрытии IBDatabase к нему "прицеплены" активные транзакции, то они завершаются как раз по TimeoutAction/DefaultAction, а не как Active:=False. Соответственно, при таком завершении (IBDatabase.Close, IBTransaction.Free) приложения в IBX изменения, произведенные в незакрытых к этому моменту транзакциях сохранятся, а в FIBPlus – исчезнут. Умолчательное поведение обоих наборов компонент может быть нежелательным в различных случаях.

3. Транзакция – это блок логической работы (последовательность операторов), который либо переводит базу данных из одного целостного состояния, либо оставляет в исходном состоянии. То есть, между StartTransaction и Commit/Rollback может быть любое число команд SQL, но оно должно быть осмысленным и взаимосвязанным.

Самый типичный пример транзакций – перевод денег с одного счета на другой:

```
StartTransaction;
```

```
update accounts set ac1 = ac1 - 100 where user_id = :x;  
update accounts set ac2 = ac2 + 100 where user_id = :x;
```

```
Commit;
```

Здесь первый оператор снимает определенную сумму с одного счета, а второй – прибавляет эту сумму к другому счету. Если бы не было транзакций, то при сбое после выполнения 1-го оператора у некоего пользователя пропали бы 100 единиц в денежном эквиваленте. Или, если после первого оператора другая программа успевает выполнить подсчет баланса (до выполнения второго update), она бы увидела отсутствие этих 100 единиц.

При работе с транзакциями все нормально – ни одна другая конкурирующая транзакция (приложение) не увидит изменений (перевода денег с одного на другой счет), пока "переводящая" транзакция не завершится. Более того, если она

завершится по Rollback, база данных (и состояние счетов пользователя) останется нетронутым.

Итак, вы сами определяете, что выполняется в транзакции. Тут все зависит от требований прикладной задачи. Допустим, вы решаете выписку накладной оформить в одной транзакции. Это хорошо. Однако, если позиций в накладной может быть много, и примерно "посередине" оформления накладной произойдет сбой приложения (сети, сервера и т. п.), то оператору придется все начинать сначала. Появление требования к сохранности промежуточного ввода может потребовать отказаться от оформления всей накладной в одной длинной транзакции. Может получиться, что выгоднее с этой точки зрения каждый элемент накладной вводить и сохранять в отдельной короткой транзакции.

Точно так же к использованию коротких транзакций вас может подвигнуть организация "списывания" товара (путем update таблицы товаров) при его выписке в накладной (триггером на insert/update/delete). Если выписка одной накладной занимает некоторое время, то возрастет вероятность конфликта update одной и той же записи товара. Следовательно, один оператор будет ждать другого, а это в 99% случаев неприемлемо. В результате придется или отказаться от оперативного снятия товара со склада, либо от длинных транзакций.

4. Транзакции должны быть максимально короткими, независимо от условий задачи. Вы можете спросить – а как же просмотр данных, или ввод большой "карточки"? Здесь есть два исключения – в Firebird с 1.0 и InterBase с 6.0 транзакции **read read_committed rec_version** – они стартуют сразу в "завершенном" состоянии, поэтому могут длиться вечно (без необходимости commit/rollback и без влияния на накопление или сборку мусора). И второе – транзакции snapshot, которые обычно используются для формирования отчетов. Но, в любом случае, стартовать транзакцию при открытии формы, а потом ждать действий пользователя, который на самом деле вполне мог в этот момент уйти на обед – плохо для версионного сервера.

То есть, в версионном сервере с длительностью транзакций легче, чем в блокировочном. Например, "короткой" можно считать транзакцию длительностью и в пол-часа. Но, здесь все зависит от активности конкурирующих транзакций. То есть, длительность транзакции – понятие относительное. И в версионном сервере она влияет не на блокировку записей (в отношении update/delete – разумеется да, но не по чтению), а на количество накапливаемых версий записей, которые могли бы стать "мусором", если бы транзакция была короче.

Конечно, к длительной работе с транзакциями подвигают стандартные Data Controls компоненты вроде DBEdit. Для того чтобы редактировать данные вы должны работать с открытым DataSet. А он может быть открыт только во время работы транзакции (использование ClientDataSet или CachedUpdates пока опустим). Многие разработчики уже поняли этот недостаток, и действуют не по следующей схеме:

```
StartTransaction;  
IBDataSet.Edit; // вход в режим редактирования
```

ожидание ввода пользователя

```
IBDataSet.Post; // отправка результатов редактирования на сервер  
Commit;
```

а по другой:

получаем данные из DataSet, прицепленного к читающей транзакции ожидаем ввод пользователя в компонентах TEdit и т. п. по нажатию кнопки "Сохранить"

```
StartTransaction;  
IBDataSet.Edit;  
// заполняем столбцы или параметры запроса данными из обычных контролов  
IBDataSet.Post;  
Commit;
```

Как видите, во втором случае длительность транзакции минимальна, т. к. она не зависит от того, сколько минут или часов пользователь набивает изменения одним пальцем на клавиатуре. Конечно, настоящие разработчики даже `IBDataSet.Edit` не вызывают. Вместо этого они используют отдельный компонент `IBSQL` с запросом `update`, и заполняют параметры запроса из контролов редактирования данных. Это дает возможность избежать головной боли по поводу "редактируемых запросов", а также гарантирует полный контроль над тем, что происходит в программе. Больше кода? Увы, да. Но если вы делаете нечто сложнее примера "Телефонная Книга" или курсовой, вам придется использовать такой подход.

5. Транзакций должно быть столько, сколько нужно в данный момент. Самый безобразный пример – в поставке Delphi, `IBMastApp`. Там всего один компонент `IBTransaction` на все приложение. Кстати, возможно некоторых смутила фраза "...прицепленного к читающей транзакции" в предыдущем примере. Ничего сложного здесь нет. Операторов SQL для работы с данными всего 5, и делятся они на две группы – которые читают данные, и которые модифицируют. Соответственно, можно взять два компонента `IBTransaction`, положить их на форму, и к одному прицеплять те `IBDataSet`, которые показывают данные в гридах, лукапах, комбобоксах и т. п., а ко второму – которые модифицируют данные (например `IBSQL/IBQuery` с операторами `insert/update/delete`).

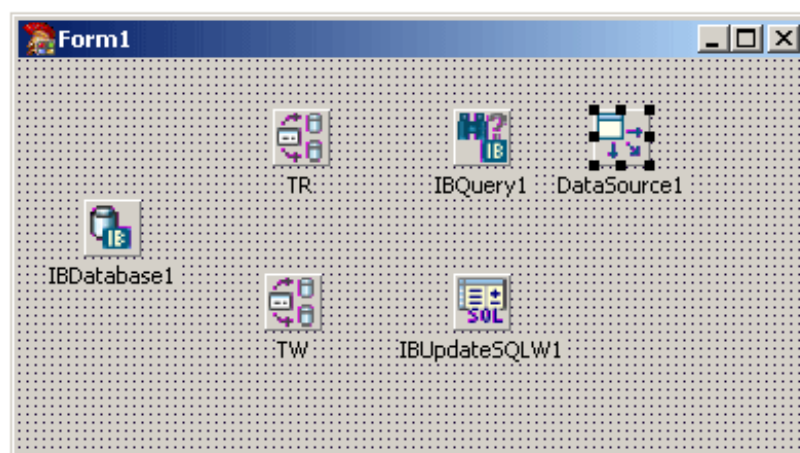
Отсюда следует, что по человечески с компонентом `IBDataSet` (а также с упомянутой ранее комбинацией `IBQuery + IBUpdateSQL`) в одной транзакции работать не получится. И действительно – к `IBDataSet` можно прицепить только одну транзакцию. Но `IBDataSet` может одновременно как читать так и редактировать данные. А для сохранения изменений данных придется делать `Commit`. А выполнение `Commit` приведет к тому, что `IBDataSet` будет автоматически закрыт, значит для удобства пользователя его надо снова открыть (а значит еще

раз выполнится запрос на чтение данных, что нагрузит сервер и сеть), и спозиционировать на ту запись, на которой до сохранения данных (и Commit) находился пользователь.

Эта проблема решается, причем один из способов описан выше – это вынос запросов insert/update/delete в отдельные компоненты IBSQL/IBQuery. Второй способ – использование IBQuery + специальная модификация IBUpdateSQL (см. IBUpdateSQLW выше), которая может быть "прицеплена" к другой транзакции, нежели IBQuery (В FIBPlus компонент pFIBDataSet по умолчанию имеет 2 свойства для подключения читающей и пишущей транзакции). Таким образом, получается, что мы можем выполнять чтение (IBQuery) в одной транзакции, а модификацию (IBUpdateSQL) – в другой. И эту другую, пишущую транзакцию, можно сделать максимально короткой – в самом простом случае в ней будет выполняться всего один оператор.

Примечание. Под "читающей" и "пишущей" транзакцией имеются в виду два компонента IBTransaction, в контексте которых будут выполняться операции чтения и записи. По умолчанию стартуемая транзакция (вообще) работает в режиме read write, поэтому при таком использовании транзакций параметры у них нужно указывать явно (вообще указывать параметры транзакций желательно всегда, даже умолчательные – по крайней мере так на форме вам будет легче понять, в каком режиме будет работать транзакция. Кроме того, как было указано выше, умолчательные параметры компонента, управляющего транзакциями, отличаются у IBX и FIBPlus). Как указать параметры транзакции только для чтения (и другие) см. в этом же документе, или в общем описании параметров транзакций.

На рисунке изображен пример формы с указанными компонентами. TR и TW – читающая и пишущая транзакции, соответственно, IBQuery1 прицеплен к TR, а IBUpdateSQLW1 – к TW.



На такую читающую транзакцию можно "навесить" массу других читающих IBDataSet/IBQuery (на картинке компоненты лежат на форме, но они могут быть и в DataModule). Например, в трехзвенных приложениях такую транзакцию можно

использовать для чтения справочников. Количество же "пишущих" транзакций не ограничено.

Примечание. По использованию транзакций в FIBPlus читайте статью www.devrace.com/ru/fibplus/articles/2165.php.

CommitRetaining?

Выше не случайно упомянут пример IBMastApp. Также тема завершения транзакций подробно изложена [в документе](#). Кроме этого, если использовать компонент FIBPlus.pFIBDataSet с двумя разными транзакциями, пишущая транзакция по умолчанию выполняет CommitRetaining сразу после выполнения операторов (действий) Insert/Update/Delete. Это поведение неверно.

Внимание! Зависит от версии FIBPlus. Обязательно проверьте поведение вашей версии при помощи компонента SQLMonitor, и только потом следуйте указанному здесь совету по настройкам timeOutAction. Также см. [статью по управлению транзакциями в FIBPlus](#).).

Короткую транзакцию не нужно завершать CommitRetaining. На самом деле CommitRetaining просто продлевает жизнь транзакции, а соответственно из "короткой" превращает ее в длинную. То есть, для сервера завершение по CommitRetaining будет сохранять все те побочные эффекты, которые вызываются длинными транзакциями.

Для того, чтобы избавиться от такого поведения pFIBDataSet нужно свойство TimeOutAction пишущей транзакции установить в taCommit (по умолчанию taRollback) – после чего автоматическое завершение этой транзакции будет происходить по Commit, а не по CommitRetaining.

В IBX такой проблемы нет, т. к. упомянутый компонент IBUpdateSQLW при AutoCommit выполняет вызов Commit для связанной с ним транзакции.

Обработка исключений и транзакции

В транзакциях обычно выполняется несколько операторов, и у разработчика может возникнуть вопрос – как правильно писать код, если выполнение одного из таких операторов может привести к ошибке?

Здесь может быть два случая:

1. старт транзакции, выполнение операторов и завершение транзакции "разбросано" по коду. Например, когда старт транзакции производится при открытии формы, затем выполняются интерактивные действия, а затем опять же по нажатию кнопки выбирается применение или отмена действий. Этот случай мы не будем рассматривать, т. к. тут все достаточно просто.

2. старт, операторы и завершение транзакции выполняется в одном блоке кода, например в процедуре обработчика нажатия кнопки.

Второй случай в коде может выглядеть так:

```
TW.StartTransaction;
```

```
try
```

```
Q1.ExecQuery;
```

```
Q2.ExecQuery;
```

```
...
```

```
TW.Commit;
```

```
except
```

```
TW.Rollback; ... // здесь может быть или raise, или сообщение пользователю об отмене
```

```
// операции в результате ошибки. Учитывайте, что это всего-лишь пример,  
// и в реальном приложении обработка except может быть куда сложнее,  
// с обработкой типа ошибки и т.п.
```

```
end;
```

Обрамлять StartTransaction в блок try особого смысла нет. Транзакция или стартует, или нет, соответственно, если при StartTransaction произошла ошибка, то вызов Commit в данном блоке не произойдет.

Иногда, например, если вы вызываете операторы DDL, например alter table drop constraint, drop index и т. п., при выполнении Commit может возникнуть ошибка – ряд операторов действует именно по Commit, а не во время выполнения оператора. Таким образом, единственным вариантом завершения транзакции в этом случае получается явный вызов Rollback. Приведенный код, с Commit "рядом" с выполнением операторов, корректно обработает данную ситуацию.

Если кроме Q1 и Q2 есть другие действия, которые также могут вызвать ошибку, или просто хочется пользователю показать ошибку правильно и на русском языке, то тогда нужно использовать обработку конкретного exception

```
except
```

```
on E: Exception do
```

```
...
```

и обращаться или к общему E.Message, или проверять E на класс ошибки EIBError.

Если же в блоке try/except вызывается только один оператор SQL (и это не оператор DDL, о чем было сказано выше), то можно обойтись без Rollback:

```
TW.StartTransaction;  
try  
  
Q1.ExecQuery;  
  
finally  
  
TW.Commit;  
  
end;
```

Дело в том, что если оператор не выполнен, то никаких изменений на сервере не произошло (кроме случая вызова процедуры с for select, внутри которого при suspend происходят изменения данных). Поэтому даже если вы вызовете Rollback, сервер автоматически превратит такой Rollback в Commit. Разумеется, если вы впоследствии допишете в блоке try/finally выполнение других операторов SQL, то такой код приведет к безусловному сохранению тех изменений, которые выполнились успешно. Сервер разрешает вызвать Commit независимо от того, были ошибки при выполнении операторов с момента StartTransaction, или нет. То есть, необходимость вызова Rollback определяется только вашим приложением и целостностью блока операторов (с логической точки зрения) внутри транзакции.

Связывание транзакций и компонент данных

Пока разработчик кладет компоненты IBX на форму, вроде бы все нормально – он сам связывает компоненты IBTransaction с IBDataSet, IBQuery, IBSQL, IBTable. Иногда разработчик забывает это сделать, в результате чего в приложении возникают длительные транзакции, "не сохраняются данные", данные "не видны" и т. п.

Такие же ошибки могут возникать, и возникают чаще при создании экземпляров компонент в коде. Например, в одном случае разработчик динамически создавал IBTransaction и IBQuery, выполнял запрос и ... где-то в коде потерял завершение транзакции. IBQuery занимался только чтением данных, поэтому ошибка проявлялась где-то через пару часов и выглядела невразумительно – заканчивались "какие-то хэндлы".

Просмотр состояния сервера (а это был IB 7.1 с временными системными таблицами), показал что количество незакрытых транзакций растет постоянно (tmp\$transactions), и так было идентифицировано наличие ошибки. Затем, после часа копания в коде, ошибка была обнаружена и исправлена.

Также бывает, когда разработчик до использования IBX имел дело с BDE, где всего одна транзакция на коннект, или с другими серверами, где тоже не бывает более одной транзакции на соединение с БД.

Клиентская часть InterBase и Firebird может стартовать какое угодно число транзакций одновременно, и выполнять их "параллельно". Поэтому компоненты получения данных с сервера должны быть обязательно привязаны к конкретной транзакции. Например, если мы выполняем код

```
TR1:=TIBTransaction.Create...
Q:=TIBQuery.Create...
TR1.StartTransaction;
Q.SQL.Clear;
Q.SQL.Add('SELECT ...');
Q.ExecQuery;
TR1.Commit;
```

то

1. выполнение Q.ExecQuery может привести к ошибке 'transaction is not active', если используемые компоненты не стартуют транзакции автоматически при выполнении запросов (и это хорошо)
2. Q.ExecQuery может выполняться в DefaultTransaction, которая создается автоматически для IBDatabase.

Как результат, запрос Q может выполняться вовсе не в транзакции TR1.

Значит, мы забыли выполнить код

```
Q.Transaction:=TR1;
```

Добавить тут практически нечего, вывод один – создаваемые динамически компоненты должны быть "привязаны" к транзакции, и привязка должна производиться явно в коде "рядом", так чтобы вы всегда могли видеть что же за транзакция используется для компонента с запросом.

Есть еще вариант, когда вместо явного TR1.StartTransaction вызывают Q.Transaction.StartTransaction. Это делается только тогда, когда присвоение Q.Transaction находится где-то вне конкретного участка кода (который вызывает Q.ExecQuery и т. д). Впрочем, вышеприведенный код можно было бы написать и так:

```
TR1:=TIBTransaction.Create...
Q:=TIBQuery.Create...
Q.Transaction:=TR1;
Q.Transaction.StartTransaction;
Q.SQL.Clear;
```

```
Q.SQL.Add('SELECT ...');  
Q.ExecQuery;  
Q.Transaction.Commit;
```

Правда, на мой взгляд это менее "явное" использование транзакций, чем прямой вызов StartTransation/Commit у именованной переменной IBTransaction.

Число транзакций в сутки

Для мониторинга транзакций (и других параметров статистики) существует инструмент IBAnalyst. Он автоматически подсчитывает среднее число стартуемых вашими приложениями транзакций для базы данных в сутки, и показывает предупреждения если, например, число активных транзакций выше 30% от среднего за день (другими словами, такие симптомы означают, что ваши приложения содержат транзакции, активные по нескольку часов). Также в меню Сервис есть Калькулятор транзакций, где вы можете определить сколько транзакций в секунду стартует 1 приложение на 1-ом рабочем месте. Исходя из этой информации вам решать – насколько интенсивно работают с транзакциями ваши приложения.

Например, в среднем в день – 100 тысяч транзакций. Рабочий день 8 часов, и одновременно работают 30 пользователей. Калькулятор выдает – 3 транзакции в секунду происходит в БД, причем в среднем 1 рабочее место стартует очередную транзакцию каждые 10 секунд. Если сопоставить это с действиями пользователя в приложении, то может оказаться, что старт транзакций в такой системе происходит чаще, чем требуется. (Например, у вас вышло, что одно рабочее место стартует 3 транзакции в секунду. Много это или мало? А вы представьте себе человека, который работает с вашей программой. Он разве похож на робота, который оформляет по 3 документа в секунду? В одной системе, где разработчик сделал выполнение каждого действия оператора в одной транзакции (тоже эдакая крайность), получилось 1500 транзакций в час на 1 рабочее место. То есть, 0.5 транзакций в секунду. Как видите, даже в этом случае число транзакций в секунду меньше.).

В промышленных системах со средней нагрузкой (до 100-150 пользователей) нормальное число транзакций в сутки (при явном управлении транзакциями) может составлять от 90 до 250 тысяч (в отдельных случаях от 400 тысяч до миллиона). Само по себе число стартуемых и завершаемых транзакций не оказывает влияния на производительность (в одной системе я наблюдал из-за ошибки разработчика по 6 миллионов транзакций в сутки, а в другой – из-за особенности ПО, автоматически вставляющего записи в БД, и делающего это в режиме "по одной транзакции на запись" – 3.5 миллиона транзакций в сутки. В обоих случаях ухудшения производительности никто и не заметил.). Ухудшить производительность может только накапливаемый в базе данных мусор, а он то

как раз и "растет", когда транзакции активны длительное время (версии записей удерживаются активными транзакциями от перехода из актуального состояния в состояние "мусор"). Понятно, что если приложение, написанное на BDE, внезапно начало сильно "тормозить" при увеличении числа пользователей, то сделать здесь уже ничего нельзя. Конечно, если используется InterBase 7.x, то можно "отловить" приложения, в которых транзакции активны длительное время, при помощи временных системных таблиц (tmp\$transactions, tmp\$attachments). Но это может иметь смысл только в редких случаях. А если сами приложения функционируют именно так, с долго работающими транзакциями, то исправить поведение приложений извне невозможно.

То есть, проблемы с транзакциями на сервере – это проблемы приложений, и только.

Что еще читать по транзакциям?

Обязательными к прочтению (если вы собираетесь писать качественные приложения, работающие с InterBase или Firebird) являются:

ClientDataSet

В отличие от "живого" просмотра и редактирования данных при помощи IBDataSet+TDataSource, данный компонент предназначен в основном для трехзвенных систем с кэшированием данных, пулами коннектов и т. п. спецификой. Также ClientDataSet используется как источник данных для Data Controls в тех случаях, когда библиотека доступа к серверу не содержит компонент, аналогичных DataSet. Это, например, [dbExpress](#). По различным аспектам работы с ClientDataSet есть [много статей](#), в данном контексте важно другое – ClientDataSet считывает все записи запроса, после чего их можно просматривать не только завершив транзакцию, но и вообще отсоединившись от сервера БД (пример BriefCase в поставке Delphi, C++Builder). Таким образом, даже длительность читающей транзакции можно сократить до минимума.

Перечитывание и сортировка записей

Для начала, примем за утверждение, что пользователь программы практически всегда хочет видеть данные (записи) упорядоченные каким-либо образом. Также ему может потребоваться периодически смотреть данные, отсортированные в том или ином виде (например список сотрудников по фамилии, по адресу, по дате рождения...). Если работать с IBDataSet/IBQuery, то это означает, что например по OnColumnClick в гриде надо сформировать запрос заново (буквально, заменить последнюю строку ORDER BY CLICKED_FIELD), и перевыполнить его. В результате при каждой такой "пересортировке" запрос отправляется на сервер, сервер выполняет запрос и сортирует данные (или выбирает их в порядке индекса), и данные передаются от сервера клиенту. Если речь идет о сортировке больших

объемов данных (десятки тысяч записей), то все это порождает как повышенную загрузку сервера, так и повышенный трафик в сети.

Решить данную проблему можно двумя способами.

Первый способ основан на ограничении выбираемых записей путем предварительной фильтрации. Например, список сотрудников сначала ограничивается путем выбора начальных букв фамилии, адреса и т. п., потом выполняется запрос. Это улучшает производительность системы, но вызывает стенания пользователей по поводу "увеличения количества действий для получения данных".

Второй способ основан на локальной буферизации выборки, и сортировки при помощи ClientDataSet. Этот компонент позволяет не только сортировать данные, но и индексировать столбцы в памяти, что повышает скорость локальной обработки данных. Если данные на сервере меняются не столь часто, или по крайней мере те данные, которые положено видеть пользователю, не меняются часто, то это – наилучший выход.

На первом этапе ClientDataSet загрузит все данные, что вызовет трафик в сети, но сам запрос может быть без указания сортировки (`select * from table`), что существенно упростит работу сервера. После чего сортировки, поиски и т. п. можно выполнять уже над ClientDataSet, который все эти операции будет производить локально, без обращения к серверу. Но, нужно помнить, что клиентский компьютер должен иметь достаточно памяти, если выполняемый запрос вернет в ClientDataSet десятки и сотни тысяч записей.

Двухфазный коммит

Смотрите описание в www.ibase.ru/ibtrans/.

Примеры

В поставке IBX (как отдельно, так и вместе с Delphi), идут примеры. Если у вас этих примеров нет, значит, версия IBX слишком старая, и надо брать обновление.

Примеры находятся тут:

- Delphi/Demos/DB/IBX или
- Delphi2005/Demos/DelphiWin32/VCLWin32/DB/IBX

Название	Описание
----------	----------

Название	Описание
<u>Admin</u>	Пример использования компонент Services API, с палитры InterBase Admin. Управление пользователями, сертификатами (лицензиями), backup/restore, статистика, свойства сервера.
CachedUp	Стандартный пример, переведенный на IBX.
IBSilentInstall	Пример использования ibinstall – библиотеки функций для создания собственных инсталляций InterBase 7.x.
IBXEvents	Стандартный пример работы с событиями, переведенный на IBX.
SQLMonitor	Пример использования компонента IBSQLMonitor, трассировки операций (запросов, транзакций и т. п.), выполняемых приложением.
ThreadedIBX	Пример параллельной работы с соединениями из разных Threads. Как обычно, в соединении с БД указан локальный коннект, который не работает параллельно. Поэтому для нормальной работы примера добавьте к IBDatabase.DatabaseName имя сервера, например localhost: (в итоге должно получиться localhost:C:\Program Files\InterBase Corp\InterBase\Examples\Database\employee.gdb).

Выше на один каталог находится застарелый пример IBMastApp, в котором на все приложение есть только один компонент IBTransaction, да еще постоянно вызывается CommitRetaining. То есть, этот пример является примером, как не надо работать с транзакциями в IBX.

Другие примеры есть там же, где находятся все дистрибутивы и обновления IBX:

Остальные примеры те же самые, что упомянуты выше по тексту статьи.