FIBPlus 6.4 Руководство разработчика

Оглавление

Соединение с базой данных

Для соединения с базой данных (БД) используйте компонент TpFIBDatabase. Подробное описание всех свойств и методов компонента можно прочитать в Приложении.

Параметры соединения

Параметры соединения являются типичными для сервера InterBase/Firebird:

- путь к файлу базы данных;
- имя пользователя и пароль;
- роль пользователя;
- набор символов;
- диалект;
- клиентская библиотека (gds32.dll для InterBase и fbclient.dll для Firebird).

Для задания всех свойств сразу удобно пользоваться встроенным диалогом настройки подключения, представленном на рисунке 1. Диалог «Database Editor» можно вызывать из контекстного меню компонента в design-time. Здесь вы можете указать необходимые параметры, взять параметры из данных псевдонима (Alias) или сохранить их в псевдониме. Также можно проверить правильность параметров, попытавшись произвести тестовое подключение.

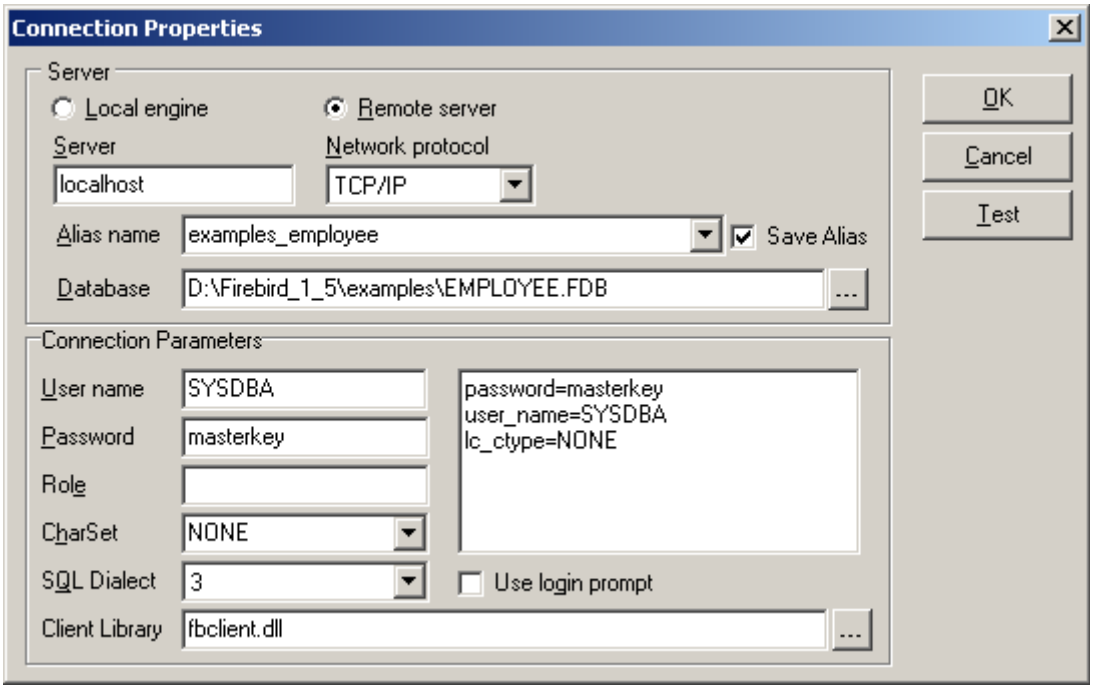

Рисунок 1. Диалог Connection Properties TpFIBDatabase

Все, что можно сделать в описанном выше диалоге, можно также сделать непосредственно из кода приложения.

Для соединения с БД нужно вызвать метод Open, либо установить свойство Connected в True. Можно использовать следующий код для подключения к базе данных:

```
function Login(DataBase: TpFIBDatabase; dbpath, uname, upass, urole: string):
Boolean;

begin
  if DataBase.Connected then DataBase.Connected := False;
  with FDataBase.ConnectParams do begin
    UserName := uname;
    Password := upass;
    RoleName := urole;
  end;
  DataBase.DBName  := dbpath;
  try DataBase.Connected := True;
  except
    on e: Exception do
      ShowMessage(e.Message);
  end;
  Result := DataBase.Connected;
end;
```

Для завершения подключения необходимо вызвать метод Close, либо установить свойство Connected в False. При этом можно централизованно закрыть все наборы данных и ассоциированные с этим подключением транзакции:

```
procedure Logout(DataBase: TpFIBDatabase);
var i: Integer;
begin
  if not DataBase.Connected then
    Exit;
  for i := 0 to DataBase.TransactionCount - 1 do
    if TpFIBTransaction(DataBase.Transactions[i]).InTransaction then
      TpFIBTransaction(DataBase.Transactions[i]).Rollback
  DataBase.CloseDataSets;
  DataBase.Close;
  end;
```

Создание и удаление БД

Создать новую базу данных очень просто. Для этого вы должны задать параметры создаваемой БД и вызвать метод CreateDatabase:

Delphi

```
with Database1 do begin
  DBParams.Clear;
  DBParams.Add('USER ''SYSDBA'' PASSWORD ''masterkey''');
  DBParams.Add('PAGE_SIZE = 2048');
  DBParams.Add('DEFAULT CHARACTER SET WIN1251');
  DBName := 'SERV_DB:C:\DB\TEST.IB';
  SQLDialect := 3;
end;

try
  Database1.CreateDataBase;
except
// Error handling
end;
```

C++

```
Database1->DBParams->Clear();
Database1->DBParams->Add("USER 'SYSDBA' PASSWORD 'masterkey'");
Database1->DBParams->Add("PAGE_SIZE = 2048");
Database1->DBParams->Add("DEFAULT CHARACTER SET WIN1251");
Database1->DBName = "SERV_DB:C:\\DB\\TEST.GDB";
Database1->SQLDialect = 3;

try
```

```
  { Database1->CreateDatabase(); }
catch (...)
{ // Error
}
```

Для удаления БД используйте метод DropDatabase. В момент использования этого метода, вы должны быть подключены к БД.

Кэширование метаданных

FIBPlus предоставляет нам возможность автоматически получать системную информацию о полях таблиц, самостоятельно настраивая такие свойства полей в TpFIBDataSet как Required (для NOT NULL полей), ReadOnly (для вычислимых полей) и DefaultExpression (для полей, у которых в базе данных задано значение по умолчанию). Это удобно и для программиста, и для пользователя, поскольку первому не нужно заботиться о ручной настройке указанных свойств во время разработки клиентского приложения, а второй получает более осмысленные сообщения при работе с программой. В частности, если какое-то поле описано в базе данных как NOT NULL, то при попытке оставить его пустым пользователь получит сообщение «Field '…' must have a value.», что гораздо проще для понимания, чем системная ошибка InterBase/Firebird о нарушении PRIMARY KEY. То же самое касается вычислимых полей, поскольку очевидно, что такие поля нельзя редактировать. FIBPlus автоматически задаст у таких полей свойство ReadOnly равное True, и пользователь не будет мучаться из-за непонятных ошибок при попытке исправить значения этих полей в TDBGrid.

Однако данная возможность FIBPlus имеет и свой недостаток, который очевидным образом проявляется при работе с низкоскоростными каналами связи. Чтобы получить информацию о полях, компоненты FIBPlus выполняют дополнительные «внутренние» запросы, обращаясь к системным таблицам InterBase/Firebird. Разумеется, при большом количестве таблиц в приложении, а также при большом количестве полей в этих таблицах, работа приложения может замедлиться, а трафик возрасти. Особенно это видно на стадии первоначальных открытий запросов, так как каждый из них сопровождается серией дополнительных. Потом, в процессе работы программы, при повторных запросах, FIBPlus использует уже полученную ранее информацию, однако при старте приложения можно обратить внимание на некоторое замедление работы.

TpFIBDatabase позволяет сохранить информацию о метаданных на клиентском компьютере и использовать ее при следующих сеансах работы. Отвечает за это свойство *TCacheSchemaOptions.*

```
TCacheSchemaOptions = class(TPersistent)
  property LocalCacheFile: string;
  property AutoSaveToFile: Boolean .. default False;
  property AutoLoadFromFile: Boolean .. default False;
  property ValidateAfterLoad: Boolean .. default True;
end;
```

Свойство *LocalCacheFile* позволяет задать имя файла, в котором будет сохраняться эта информация. *AutoSaveToFile* отвечает за автоматическую запись кеша в файл при закрытии приложения. *AutoLoadFromFile* отвечает за загрузку кеша из файла. И, наконец, *ValidateAfterLoad* указывает на то, стоит ли проверять сохраненный кеш после загрузки. В дополнение к этим свойствам добавляется событие *OnAcceptCacheSchema*, где вы можете указать для каких объектов не нужно загружать сохраненную информацию.

Использовать это свойство также очень просто.

```
with pFIBDatabase1.CacheSchemaOptions do begin
  LocalCacheFile := 'fibplus.cache';
  AutoSaveToFile := True;
  AutoLoadFromFile := True;
  ValidateAfterLoad:= True;
end;
```

Изменение метаданных в процессе работы клиентских приложений может привести к ошибкам в работе клиентов. Чтобы избежать появления этих ошибок можно например сделать триггер на таблицы репозитариев, который выдавал бы определенное событие. По этому событию можно чистить кэш метаданных во время исполнения программы.

В модуле pFIBDataInfo содержатся классовые переменные на каждый тип кэшируемой информации. Для того чтобы очистить интересующую вас информацию нужно выполнить метод Clear. Также некоторые компоненты имеют методы для точечной очистки кеша, например для определенной таблицы или определенного TpFIBDataSet или даже по коду DataSet_ID из репозитория.

```
ListTableInfo  :TpFIBTableInfoCollect;
ListDataSetInfo:TpDataSetInfoCollect;
ListSPInfo     :TpStoredProcCollect;
ListErrorMessages:TpErrorMessagesCollect;
```

Кэширование блоб-полей

Кэширование блоб-полей на клиентской стороне – это еще одна уникальная особенность FIBPlus. При включении BlobSwapSupport.Active := True, FIBPlus будет автоматически сохранять полученные BLOB-поля в указанном каталоге (свойство SwapDir). По умолчанию свойство SwapDir принимает значение равное {APP_PATH}, то есть, равное каталогу, в котором находится исполняемое приложение. При необходимости, можно указать также подкаталог, в котором будут сохраняться BLOB-поля. Например, SwapDir := '{APP_PATH}' + '\BLOB_FILES\'

В компоненте TpFIBDatabase для поддержки работы с данным свойством существуют 4 события:

```
property BeforeSaveBlobToSwap: TBeforeSaveBlobToSwap;
property AfterSaveBlobToSwap: TAfterSaveLoadBlobSwap;
property AfterLoadBlobFromSwap: TAfterSaveLoadBlobSwap;
property BeforeLoadBlobFromSwap: TBeforeLoadBlobFromSwap;
```

где

```
TBeforeSaveBlobToSwap = procedure(const TableName, FieldName: string;
RecordKeyValues: array of variant; Stream: TStream; var FileName: string; var
CanSave: boolean) of object;
TAfterSaveLoadBlobSwap = procedure(const TableName, FieldName: string;
RecordKeyValues: array of variant; const FileName: string) of object;
TBeforeLoadBlobFromSwap = procedure(const TableName, FieldName: string;
RecordKeyValues: array of variant; var FileName: string; var CanLoad: boolean)
of object;
```

Обработчики служат для более гибкого управления процессом сохранения и чтения BLOB-полей с диска. В частности, в обработчиках перед сохранением BLOB-поля можно запретить сохранение конкретного BLOB-поля в зависимости от имени таблицы и поля, значений других полей записи, свободного места на диске и т.д. Регулировать процесс

сохранения можно и при помощи свойства MinBlobSizeToSwap, в котором можно задать минимальный размер BLOB-полей, которые будут сохраняться на диске.

Технология имеет ряд ограничений:

1. Таблица должна иметь первичный ключ.
2. Чтение BLOB-полей должна производиться компонентом TpFIBDataSet.
3. Приложение само должно следить за свободным местом на диске. В частности, такую функцию можно реализовать в обработчике события BeforeSaveBlobToSwap.
4. К сожалению, после backup/restore базы данных внутри базы данных происходит замена всех BLOB_ID, а потому локальный кэш теряет актуальность и автоматически вычищается. Необходимо пояснить, что при очередном подключении вашего приложения к базе данных, FIBPlus автоматически запускает специальный поток, который в отдельном подключении проверяет весь кэш на диске на наличие соответствующих BLOB-полей в базе данных. Для отсутствующих BLOB-полей файлы сразу же удаляются.

Клиентские Блоб-фильтры

Это пользовательские функции, которые позволяют обрабатывать (кодировать/ декодировать, упаковывать и т.д.) blob-поля на клиентской стороне прозрачно для пользовательского приложения. Это может оказаться полезным, чтобы архивировать или шифровать содержимое blob-полей в базе данных, причем менять для этого клиентскую программу не нужно. В FIBPlus реализован механизм клиентских blob-фильтров, который очень похож по своей сути на механизм, встроенный в InterBase/Firebird. Механизм реализуется за счет регистрации двух процедур для чтения и записи blob-поля в TpFIBDatabase. В результате FIBPlus будет автоматически использовать данные процедуры для обработки всех blob-полей заданного типа во всех TpFIBDataSet, использующих один экземпляр TpFIBDatabase.

Чтобы использовать эту технологию нам нужно написать обработчики кодирования/декодирования для каждого типа блоб-поля и зарегистрировать эти обработчики. Помните, что типы пользовательских блоб-полей при описании их в базе данных должны быть отрицательными.

Например, если необходимо сжимать блоб-поля, то необходимо описать два метода для упаковки/распаковки:

```
procedure PackBuffer(var Buffer: PChar; var BufSize: LongInt);
var srcStream, dstStream: TStream;
begin
  srcStream := TMemoryStream.Create;
  dstStream := TMemoryStream.Create;
  try
    srcStream.WriteBuffer(Buffer^, BufSize);
    srcStream.Position := 0;
    GZipStream(srcStream, dstStream, 6);
    srcStream.Free;
    srcStream := nil;
    BufSize := dstStream.Size;
    dstStream.Position := 0;
    ReallocMem(Buffer, BufSize);
    dstStream.ReadBuffer(Buffer^, BufSize);
  finally
    if Assigned(srcStream) then srcStream.Free;
    dstStream.Free;
  end;
```

```
end;

procedure UnpackBuffer(var Buffer: PChar; var BufSize: LongInt);
var srcStream,dstStream: TStream;
begin
  srcStream := TMemoryStream.Create;
  dstStream := TMemoryStream.Create;
  try
    srcStream.WriteBuffer(Buffer^, BufSize);
    srcStream.Position := 0;
    GunZipStream(srcStream, dstStream);
    srcStream.Free;
    srcStream:=nil;
    BufSize := dstStream.Size;
    dstStream.Position := 0;
    ReallocMem(Buffer, BufSize);
    dstStream.ReadBuffer(Buffer^, BufSize);
  finally
    if assigned(srcStream) then srcStream.Free;
    dstStream.Free;
  end;
end;
```

Теперь перед соединением с БД мы должны их просто зарегистрировать. Это делается при помощи вызова функции RegisterBlobFilter. Значение первого параметра – это тип blob-поля (в нашем случае, –15), второй и третий параметры – это функции кодирования и декодирования:

pFIBDatabase1.RegisterBlobFilter(-15, @PackBuffer, @UnpackBuffer);

Использования этой технологии можно посмотреть в демонстрационном примере BlobFilters.

Обработка потери соединения

FIBPlus предоставляет своим пользователям уникальную возможность обработки потери соединения. Для обработки потери соединения используется сам компонент TpFIBDatabase и компонент централизованной обработки всех ошибок библиотеки TpFIBErrorHandler.

Пример использования этой функциональности представлен в примере ConnectionLost. Коротко опишем, как он устроен. Компонент TpFIBDatabase реализует три специальных события:

AfterRestoreConnect – возникает при успешном восстановлении соединения.

OnLostConnect – возникает при потере соединения. Событие возникает в момент очередного обращения к БД, которое завершается ошибкой. Здесь вы можете задать одно из трех возможных действий, которые можно предпринять в этой ситуации (смотрите описание `TOnLostConnectActions`) - закрыть приложение, закрыть соединение, проигнорировать, попытаться восстановить соединение.

OnErrorRestoreConnect – возникает при очередной ошибке попытки восстановления соединения.

При потере соединения пользователю предоставляется выбор, какое из действий предпринять. В случае успеха выдается сообщение, что соединение восстановлено. При очередной ошибке восстановления соединения мы можем посчитать попытки в нашем коде и предпринимать какие-то иные действия в случае необходимости.

На компоненте TpFIBErrorHandler мы остановимся в отдельном разделе, сейчас же скажем, что обработчик ошибок при возникновении потери соединения просто подавляет стандартную реакцию на ошибку.

```
procedure TForm1.dbAfterRestoreConnect(Database: TFIBDatabase);
begin
  MessageDlg('Connection restored', mtInformation, [mbOk], 0);
end;

procedure TForm1.dbErrorRestoreConnect(Database: TFIBDatabase;
  E: EFIBError; var Actions: TOnLostConnectActions);
begin
  Inc(AttemptRest);
  Label4.Caption:=IntToStr(AttemptRest);
  Label4.Refresh
end;

procedure TForm1.dbLostConnect(Database: TFIBDatabase; E: EFIBError;
  var Actions: TOnLostConnectActions);
begin
  case cmbKindOnLost.ItemIndex of
   0: begin
       Actions := laCloseConnect;
       MessageDlg('Connection lost. TpFIBDatabase will be closed!',
        mtInformation, [mbOk], 0);
      end;
   1:begin
      Actions := laTerminateApp;
      MessageDlg('Connection lost. Application will be closed!',
       mtInformation, [mbOk], 0
      );
     end;
   2:Actions := laWaitRestore;
  end;
end;

procedure TForm1.pFibErrorHandler1FIBErrorEvent(Sender: TObject;
  ErrorValue: EFIBError; KindIBError: TKindIBError; var DoRaise: Boolean);
begin
  if KindIBError = keLostConnect then begin
    DoRaise := false;
    Abort;
  end;
end;
```

Други полезные методы

Компонент TpFIBDatabase реализует множество полезных методов. Перечислим некоторые из них, которые, на наш взгляд, используются наиболее часто.

Выполнение простых SQL-запросов

Если Вам нужно выполнить какой-либо простой SQL-запрос для получения или установки каких-либо параметров необходимых для работы приложения можно воспользоваться следующими методами:

```
function Execute(const SQL: string): boolean;
```

- выполняет SQL-запрос, который передали в параметре SQL, и возвращает истину в случае успеха.

```
function QueryValue(const aSQL: string; FieldNo:integer; ParamValues:array of
variant; aTransaction:TFIBTransaction=nil):Variant; overload;
```

- получает значение поля с индексом FieldNo как результат выполнения aSQL в транзакции aTransaction. Транзакцию можно не указывать, тогда будет использована транзакция DefaultTransaction. Можно передать в запрос параметры. Значение возвращается в виде переменной типа Variant. Используя похожий метод QueryValueAsStr можно получить значение в виде строки, а при помощи QueryValues получить вариантный массив значений. Запомните, что ваш SQL в этом случае должен возвращать не более одной строки.

Получение значений генераторов

Используйте метод для получения значение генератора

```
function Gen_Id(const GeneratorName: string; Step: Int64; aTransaction:
TFIBTransaction = nil): Int64;
```

Получение информации о таблицах и полях

```
procedure GetTableNames(TableNames: TStrings; WithSystem: Boolean);
procedure GetFieldNames(const TableName: string; FieldNames: TStrings;
WithComputedFields: Boolean = True);
```

Первый метод получает имена всех таблиц и заполняет ими список TableNames. Параметр WithSystem указывает, извлекать ли имена системных таблиц.

Второй метод получает имена полей таблицы TableName и заполняет ими список FieldNames. Параметр WithComputedFields указывает, извлекать ли вычисляемые поля, описанные в БД как «COMPUTED BY».

Работа с транзакциями

Транзакция - это операция перевода БД из одного непротиворечивого состояния в другое непротиворечивое состояние.

Все действия, выполняемые при работе с БД (получение, либо изменение данных или метаданных) осуществляются в контексте некоторой транзакции. И понимание работы TpFIBTransaction является ключевым моментом для понимания тонкостей работы FIBPlus. Поэтому, мы настоятельно рекомендуем обратиться к разделу «Working with Transaction» ApiGuide.pdf документации InterBase.

Все изменения, выполненные в контексте транзакции, можно либо подтвердить - Commit (при отсутствии ошибок), либо отменить - Rollback. В дополнение к этим базовым методам в TpFIBTransaction реализованы аналогичные методы с сохранением контекста - CommitRetaining и RollbackRetaining.

Для запуска транзакции нужно вызвать метод StartTransaction компонента или установить свойство Active в True. Для подтверждения транзакции нужно вызывать метод Commit/CommitRetaing, а для отмены - метод Rollback/RollbackRetaining Такие компоненты как TpFIBQuery и TpFIBDataSet имеют свои свойства, которые позволяют не заботиться об управлении транзакциями. В частности, свойство TpFIBDataSet.AutoCommit, параметр poStartTransaction свойства TpFIBDataSet.Options, а также параметры qoStartTransaction и qoCommitTransaction в свойстве TpFIBQuery.Options.

Настройка параметров транзакции

Параметры транзакций очень серьезная тема, выходящая за рамки данного документа. Мы настоятельно рекомендуем ознакомиться с документацией по InterBase. Тем не менее, в большинстве ситуаций нет необходимости вникать во все тонкости управления транзакциями на уровне API, поэтому FIBPlus реализует ряд механизмов для упрощения работы программиста. В частности, TpFIBTransaction реализует три встроенных типа транзакции tpbDefault, tpbReadCommited, tpbRepeatableRead. Вы также можете создать свои специфические типы в design-time в редакторе компонента TpFIBTransaction (рисунок 2) и использовать их таким же образом, как и встроенные. Указав тип транзакции, вы задаете ее параметры:

tpbDefault – параметры должны быть объявлены в TRParams

tbpReadCommited – уровень изоляции ReadCommited

tbpRepeatableRead – уровень изоляции RepeatableRead

Планирование использования транзакций в приложениях

Эффективность работы InterBase в большой степени достигается правильным использованием транзакций в приложениях. Версионная архитектура такова, что обновляющие транзакции удерживают версии записей. Поэтому в общем случае нужно стараться, чтобы ваши обновляющие транзакции были как можно короче. Транзакции только для чтения могут оставаться открытыми сколько угодно, поскольку они версий не удерживают.

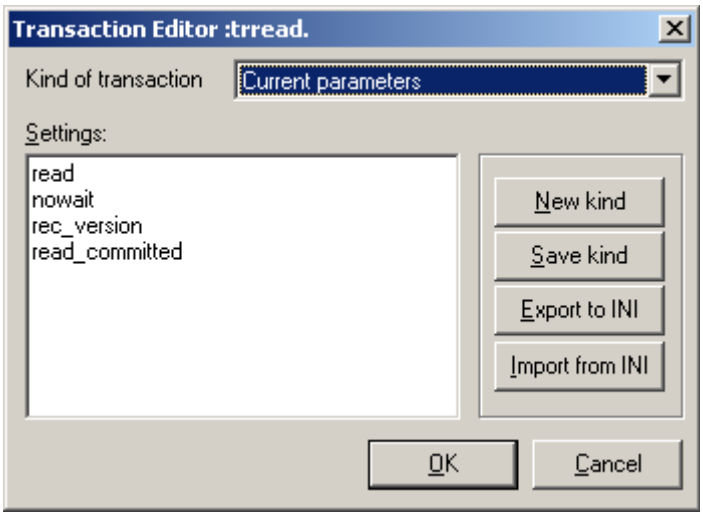

Рисунок 2. Диалог Transaction Editor

Использование SavePoints

Сервера InterBase и их клоны не поддерживают вложенных транзакций. Но InterBase 7.X и Firebird 1.5 поддерживают средства создания контрольных точек сохранения SavePoints и возврата к любой из точек. FIBPlus поддерживает эту функциональность при помощи трех методов:

```
procedure SetSavePoint(const SavePointName:string);
procedure RollBackToSavePoint(const SavePointName:string);
procedure ReleaseSavePoint(const SavePointName:string);
```

Первый метод устанавливает точку восстановления с именем SavePointName. Второй откатывает транзакцию до состояния в точке SavePointName. Третий освобождает ресурсы сервера, связанные с точкой восстановления SavePointName.

Выполнение SQL- запросов

Взаимодействие приложения с базой данных заключается в выполнении SQL-операторов. Они используются для получения и модификации данных и метаданных. В FIBPlus за выполнение SQL-операторов отвечает компонент TpFIBQuery. Это быстрый, легкий и вместе с тем мощный компонент для выполнения любых действий над БД.

Использовать компонент очень просто. Достаточно указать для него компонент TpFIBDatabase, заполнить свойство SQL и вызвать один из методов ExecQuery (ExecQueryWP, ExecQueryWPS). Следующий пример кода демонстрирует, как создать компонент TpFIBQuery динамически в run-time и получить с его помощью данные о клиентах.

```
var sql: TpFIBQuery;

sql := TpFIBQuery.Create(nil);
with sql do
try
  Database := db;
  Transaction := db.DefaultTransaction;
  SQL.Text := 'select first_name, last_name from customer';
  ExecQuery;
  while not Eof do begin
    Memo1.Lines.Add(
      FldByName['FIRST_NAME'].AsString+' '+
```

```
        FldByName['LASTST_NAME'].AsString);
      Next;
    end;
    sql.Close;
  finally
    sql.Free;
  end;
```

Передача параметров

Очень часто возникает необходимость передачи параметров в SQL запросы. В FIBPlus для этого используются свойство Params и методы ParamsCount, ParamByName компонента TpFIBQuery. Кроме того, существуют методы выполнения запроса с предварительной установкой параметров. Это семейство методов ExecWP. Передавать параметры стало действительно очень просто. Сравните несколько вариантов кода, приведенных ниже:

```
sql.SQL.Text :=
  'select first_name, last_name from customer'+
  'where first_name starting with :first_name';

{ 1 вариант }
sql.ParamByName('first_name').AsString := 'A';
sql.ExecQuery;

{ 2 вариант }
sql.ExecWP('first_name', ['A']);

{ 3 вариант }
sql.ExecWP(['A']);
```

SQL- секции

FIBPlus предоставляет вам широкие возможности по управлению текстом SQL в ваших запросах. Прежде всего, это секции SQL – список полей, условий, порядка группировки и сортировки, план выполнения запроса.

Это простые строковые свойства доступные как для чтения, так и для записи:

FieldsClause – содержит список полей;

MainWhereClause – содержит основную секцию WHERE (подробности в разделе ниже)

OrderClause – секция «order by»

GroupByClause – секция «group by»

PlanClause – план.

Также в FIBPlus реализованы такие уникальные возможности, как макросы и conditions - расширенный механизм работы с секцией where. Остановимся на них подробно.

Макросы

Механизм макросов позволяет вам управлять вариационной частью ваших запросов. Вы можете менять и уточнять их смысл без переписывания текста запроса.

Макрос выглядит следующим образом **@@<MACROS_NAME>|%<DEFAULT_VALUE>||#|@**

Т.е., макрос - это специфическая последовательность символов между знаками @@ и @. После @@ следует обязательный параметр – имя макроса. Можно также задать значение макроса по умолчанию, которое отделяется от имени символом %. Есть возможность задать

необязательный параметр #, который будет говорить FIBPlus, что значение макроса нужно заключить в кавычки.

Использование макросов очень похоже на использование параметров. Продемонстрируем это:

```
Sql.SQL.Text := 'select * from @@table_clause@ where @@where_clause% 1=1@';
Sql.ExecWP(['CUSTOMER','FIRST_NAME STARTING WITH ''A''']);
```

Для того чтобы установить значение макроса в значение по умолчанию, нужно вызвать метод SetDefaultMacroValue объекта-параметра.

Макрос может также содержать параметр. В этом случае для поиска параметра необходимо использовать функцию FindParam, а для установки параметра использовать методом ParamByName:

```
Sql.SQL.Text := 'select * from @@table_clause@ where @@where_clause% 1=1@';
Sql.Params[0].AsString := 'CUSTOMER';
Sql.Params[1].AsString := 'CUST_NO = :CUST_NO';
if Assigned(Sql.FindParam('CUST_NO')) then
  Sql.ParamByName('CUST_NO').AsInteger := 1001;
Sql.ExecQuery;
```

Использование макросов для TpFIBDataSet демонстрируется в примере ServerFilterMarcoses.

Условия

Механизм условий (conditions) - это еще одна возможность для изменения вариативной части ваших SQL-запросов.

Для любого SQL в design-time (рисунок 3) можно задать один или несколько параметров-условий. Для этого удобно пользоваться встроенным диалогом, приведенным на рисунке ....

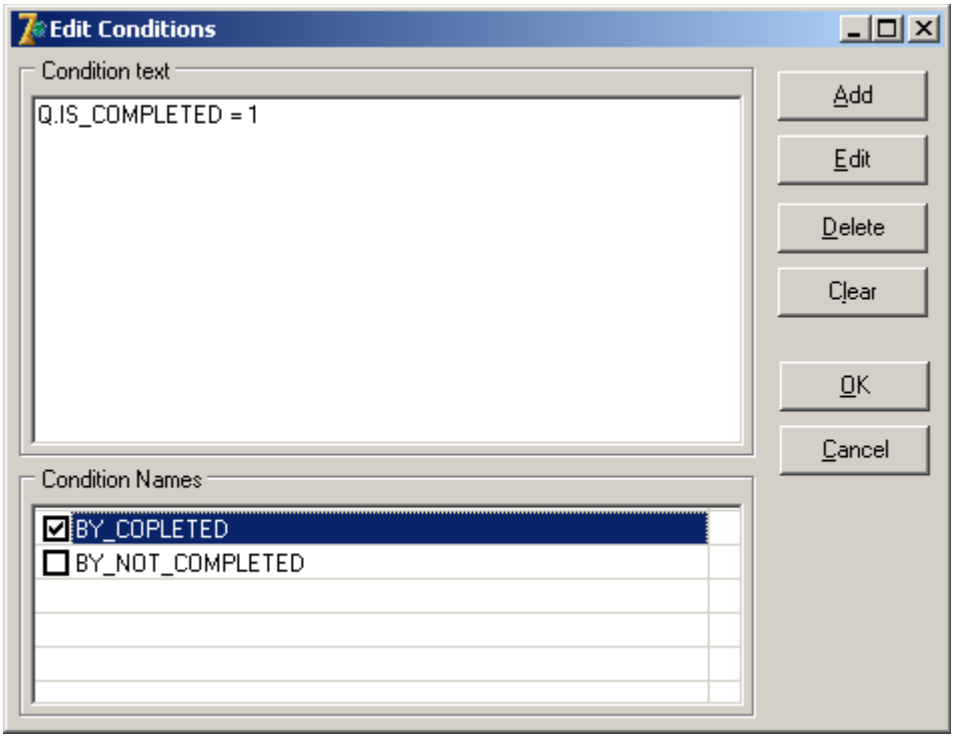

Рисунок 3. Диалог Edit Conditions.

Чтобы

включить условие достаточно установить его свойство Active в True.

pFIBQuery1.Conditions[0].Active := True;

или

pFIBQuery1.Conditions.ByName('by_customer').Active := True;

Код для работы с Conditions мог бы выглядеть следующим образом:

```
if pFIBQuery1.Open then pFIBQuery1.Close;
pFIBQuery1.Conditions.CancelApply;
pFIBQuery1.Conditions.Clear;
if byCustomerFlag then
  pFIBQuery1.Conditions.AddCondition('by_customer', 'cust_no = 1001', True);
pFIBQuery1.Conditons.Apply;
pFIBQuery1.Open;
```

В TpFIBDataSet реализованы два дополнительных метода – CancelConditions и ApplyConditions, которые вызывают, соответственно, Conditions.Cancel и Conditions.Apply. Код для TpFIBDataSet выглядит немного проще.

```
with pFIBDataSet1 do begin
  if Active then Close;
  CancelConditioins;
  Conditions.Clear;
  if byCustomerFlag then Conditions.AddCondition('by_customer', 'cust_no =
1001', True);
  ApplyConditions;
  Open;
end;
```

Использование условий для TpFIBDataSet продемонстрировано в примере ServerFilterConditions

Пакетная обработка

FIBPlus содержит встроенные методы для пакетной обработки данных, так называемые Batch-методы. Эти методы могут пригодиться для репликации между базами данных, и при операциях импорта и экспорта.

```
function BatchInput(InputObject: TFIBBatchInputStream) :boolean;
function BatchOutput(OutputObject: TFIBBatchOutputStream):boolean;
procedure BatchInputRawFile(const FileName:string);
procedure BatchOutputRawFile(const FileName:string;Version:integer=1);
procedure BatchToQuery(ToQuery:TFIBQuery; Mappings:TStrings);
```

Параметр Version введен для совместимости форматов со старыми версиями файлов, выгруженных FIBPlus при помощи метода BatchOutputXXX. Если Version = 1, то используется старый принцип, при котором во внешний файл выводятся только данные в порядке, который определяется полями SQL-запроса. Предполагается, что при чтении данных, сохраненных методом BatchInputRawFile, в читающем SQL параметры будут расположены в том же порядке. Количество полей TpFIBQuery, данные которого записывались, и количество параметров в TpFIBQuery, который потом будет считывать данные, должны совпадать. Для строковых полей необходимо, чтобы длина записываемого поля и читающего параметра также совпадали, однако совпадения имен не требуется. Если Version = 2, то используется новый принцип записи. Помимо данных в файл записывается служебная информация о полях (имя, тип и длина). При последующем чтении, данные будут читаться по принципу совпадения имен. Порядок и количество полей в

записывающем TpFIBQuery может не совпадать с порядком и количеством параметров в читающем TpFIBQuery. Типы и длина тоже могут не совпадать. Требуется лишь совпадение имен.

Работать с batch-методами очень просто. Покажем это на простом примере. Код, приведенный ниже, состоит из трех частей. Первая часть сохраняет данные о клиентах во внешний файл, а вторая загружает эти данные в БД. Третья часть показывает, как можно сделать преобразование данных.

```
{ I }
pFIBQuery1.SQL := 'select EMP_NO, FIRST_NAME, LAST_NAME from CUSOMER';
pFIBQuery1.BatchOutputRawFile('employee_buffer.fibplus', 1);

{ II }
pFIBQuery1.SQL := 'insert into employees(EMP_NO, FIRST_NAME, LAST_NAME)'+
  ' values(:EMP_NO, :FIRST_NAME, :LAST_NAME)';
pFIBQuery1.BatchInputRawFile('employee_buffer.fibplus');

{ III }
pFIBQuery1.SQL := 'select EMP_NO, FIRST_NAME, LAST_NAME from CUSOMER';
pFIBQuery2.SQL := 'insert into tmp_employees(EMP_NO, FIRST_NAME, LAST_NAME)'+
  ' values(:EMP_NO, :FIRST_NAME, :LAST_NAME)';

mapStrings.Add('EMP_NO=EMP_NO');
mapStrings.Add('FIRST_NAME=FIRST_NAME');
mapStrings.Add('LAST_NAME=LAST_NAME');

pFIBQuery1.BatchToQuery(pFIBQuery2, mapStrings);
```

Мы еще вернемся к пакетной обработке при рассмотрении TpFIBDataSet, который также содержит несколько методов пакетной обработки.

При возникновении ошибки выполнения обработки возникает событие OnBatchError. В коде этого события, используя параметр BatchErrorAction (TBatchErrorAction = (beFail, beAbort, beRetry, beIgnore)) можно принять решение о том, что делать в случае ошибки.

Выполнение хранимых процедур

Выполнение процедур, практически не отличается от выполнения запросов. Вы просто пишете в тексте SQL 'execute procedure some_proc(:proc_param)' либо, для селективных процедур, 'select * from some_proc(:proc_param)'.

Если неселективная процедура возвращает какие-либо результаты, их можно получить после выполнения запроса через уже известное нам свойство Fields.

```
Sql.SQL.Text := 'execute procedure some_proc(:proc_param)';
Sql.ExecWP([25]);
ResultField1 := Sql.Fields[0].AsInteger;
```

Это немного отличается от BDE, ADO и других компонент доступа, где выходные данные доступны также через параметры.

Кроме того, в FIBPlus хранимые процедуры можно выполнять при помощи компонента TpFIBStoredProc. Он является прямым наследником TpFIBQuery, у которого есть свойство StoredProcName и его рекомендуется использовать для выполнения неселективных процедур.

Выполнение DDL операторов.

Кроме выполнения SQL-операторов, компонент TpFIBQuery позволяет выполнять DDL операторы. Для этого необходимо установить свойство ParamsCheck в False. DDL- операции выполняются по одной и никаких разделителей не ставится. В новых версиях для DDL поддерживаются макросы.

Повторное использование запросов

Для того чтобы подготовить запрос к выполнению, все клиентские библиотеки, включая FIBPlus, передают на сервер полный текст запроса. Для выполнения же подготовленного запроса достаточно передавать на сервер только Handle и значения параметров. Если в вашей программе часто используются одинаковые запросы, то вы вполне можете организовать их повторное использование при помощи методов модуля из pFIBCacheQueries.pas:

```
function GetQueryForUse (aTransaction: TFIBTransaction; const SQLText: string):
TpFIBQuery;
procedure FreeQueryForUse (aFIBQuery: TpFIBQuery);
```

Вам не придется создавать экземпляры TpFIBQuery - процедура GetQueryForUse самостоятельно создаст его при первом вызове, а потом будет возвращать ссылку на существующий компонент, если вы будете выполнять этот же запрос снова и снова. Очевидно, что при каждом последующем вызове будет использоваться уже подготовленный к выполнению запрос, а, следовательно, передача текста запроса на сервер будет выполнена только один раз. После получения результатов запроса из компонента TpFIBQuery необходимо вызвать метод FreeQueryForUse. Данный механизм уже используется для внутренних целей в компонентах FIBPlus, в частности, при вызове генераторов для получения значений первичных ключей. Вы также можете воспользоваться этими методами в своих приложениях для оптимизации трафика.

Работа с наборами данных

И, наконец, мы подошли к работе с наборами данных. Это компонент TpFIBDataSet. Он работает на основе компонента TpFIBQuery и позволяет кэшировать результаты выборки. TpFIBDataSet является наследником TDataSet и полностью поддерживает все его свойства, события и методы. Для получения более подробной информации о TDataSet смотрите встроенную справку Delphi/C++Builder.

Базовые принципы работы с наборами данных

TpFIBDataSet позволяет получать, вставлять, изменять и удалять данные. Все операции реализованы при помощи соответствующих компонентов TpFIBQuery в его составе.

Для того чтобы выбрать данные, необходимо заполнить свойство SelectSQL. Это равнозначно заполнению свойства SQL компонента QSelect типа TpFIBQuery. Для вставки данных требуется заполнить свойство InsertSQL.Text. Аналогично задаются запросы для модификации (UpdateSQL), удаления (DeleteSQL) и обновления текущей записи (RefreshSQL).

Для примера возьмем демонстрационную базу данных employee.gdb (fdb, если это Firebird). Напишем SelectSQL для получения всех служащих, укажем запросы в InserSQL и т.д.

```
with pFIBDataSet1 do begin
  if Active then Close;
  SelectSQL.Text :=
  'select CUST_NO, CUSTOMER, CONTACT_FIRST, CONTACT_LAST from CUSTOMER';

  InsertSQL.Text :=
  'insert into CUSTOMER(CUST_NO, CUSTOMER, CONTACT_FIRST, CONTACT_LAST )'+
  ' values (:CUST_NO, :CUSTOMER, :CONTACT_FIRST, :CONTACT_LAST)';

  UpdateSQL.Text :=
  'update CUSTOMER set CUSTOMER = :CUSTOMER, '+
  'CONTACT_FIRST = :CONTACT_FIRST, CONTACT_LAST = :CONTACT_LAST '+
  'where CUST_NO = :CUST_NO';

  DeleteSQL.Text := 'delete from CUSTOMER where CUST_NO = :CUST_NO';

  RefreshSQL.Text :=
  'select CUST_NO, CUSTOMER, CONTACT_FIRST, CONTACT_LAST ' +
  'from CUSTOMER where CUST_NO = :CUST_NO';

  Open;
end;
```

Для того, чтобы открыть TpFIBDataSet, необходимо выполнить метод Open или OpenWP, или установить свойство Active в True. Чтобы закрыть TpFIBDataSet, вызовите метод Close.

Пусть вас не пугает количество написанного текста, все эти запросы сделает за вас редактор TpFIBDataSet, который можно вызвать через контекстное меню компонента. Диалог показан

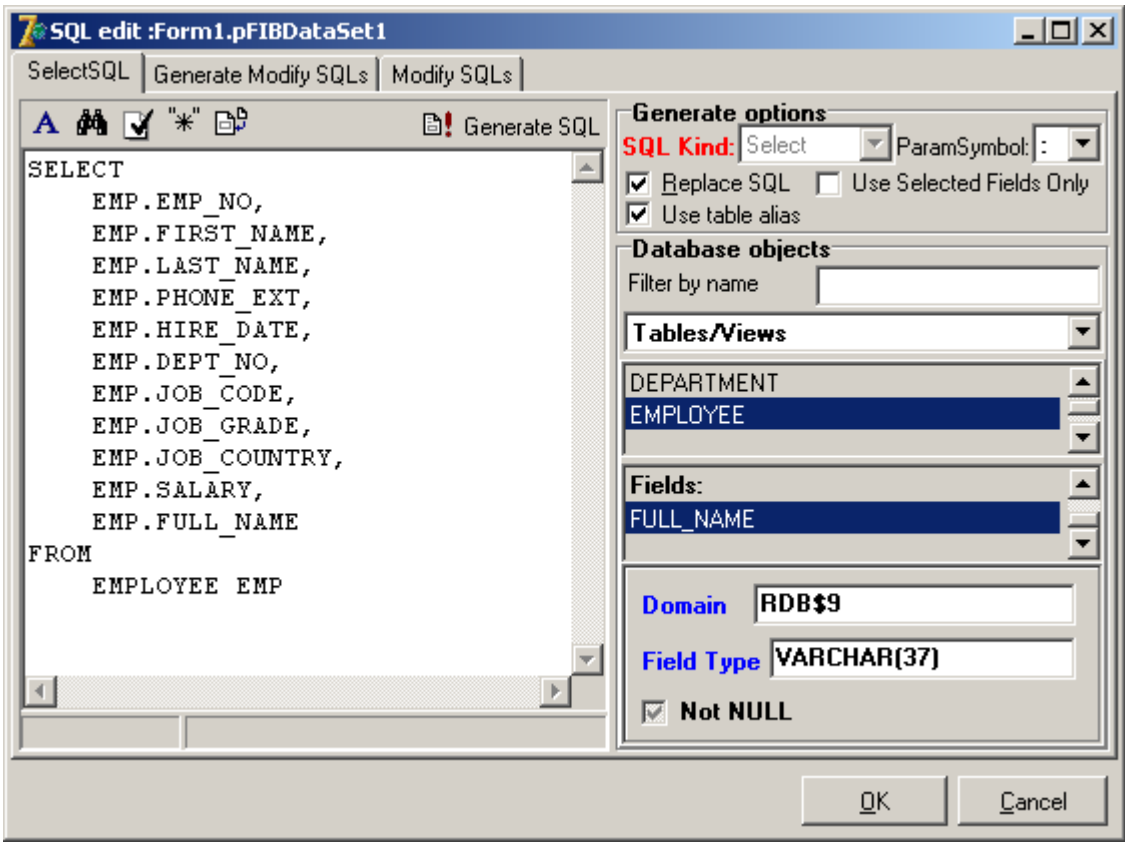

Рисунок 4. Дилалог SQL Edit TpFIBDataSet

на рисунке 4.

Простое приложение, демонстрирующее использование редактируемого TpFIBDataSet, представлено в примере DataSetBasic.

Автоматическая генерация обновляющих запросов

Кроме использования SQL-редактора TpFIBDataSet, FIBPlus может построить все обновляющие запросы в режиме исполнения программы, и сделать это возможно даже эффективнее, чем при генерации запросов в design-time.

Необходимо использовать свойство AutoUpdateOptions. Это очень важная группа настроек, которое можно значительно упростить вам жизнь.

```
TAutoUpdateOptions= class (TPersistent)
  property AutoParamsToFields: Boolean .. default False;
  property AutoReWriteSqls: Boolean .. default False;
  property CanChangeSQLs: Boolean .. default False;
  property GeneratorName: string;
  property GeneratorStep: Integer .. default 1;
  property KeyFields: string;
  property ParamsToFieldsLinks: TStrings;
  property SeparateBlobUpdate: Boolean .. default False;
  property UpdateOnlyModifiedFields: Boolean .. default False;
  property UpdateTableName: string;
  property WhenGetGenID: TWhenGetGenID .. default wgNever;
end;

TWhenGetGenID=(wgNever,wgOnNewRecord,wgBeforePost);
```

AutoRewriteSQLs при наличии пустых SQLText для InsertSQL, UpdateSQL, DeleteSQL,

RefreshSQL будет автоматически производится их генерация на основе свойств SelectSQL, KeyFields и UpdateTableName.

CanChangeSQLs говорит о том, что разрешена перезапись непустых запросов.

GeneratorName и *GeneratorStep* задают, соответственно, имя и шаг генератора.

KeyFields содержит список ключевых полей.

SeparateBlobUpdate управляет записью блоб-полей в базу данных. Если опция установлена, то будет сначала производится запись строки без блоб-поля, и, в случае успеха, будет записываться и само блоб-поле.

При установленной опции *UpdateOnlyModifiedFields*, если установлена также опция *CanChangeSQLs*, для каждой операции обновления будет автоматически формироваться новый SQL-запрос, в котором будут представлены только те поля, которые реально были изменены.

UpdateTableName должна содержать имя обновляемой таблицы

WhenGetGenId позволяет задать режим использования генератора для формаирования первичного ключа – не генерировать ключ, генерировать при добавлении новой записи, генерировать непосредственно перед операцией Post.

T.e. FIBPlus, при использовании настроек AutoUpdateOptions позволяет избавиться от генерации модифицирующих SQL в режиме дизайна и переложить эту задачу на время исполнения программы. Для этого Вам, фактически, нужно лишь прописать имя обновляемой таблицы и ключевое поле.

Все это можно как настраивать как в режиме дизайна, так и в режиме выполнения. Код взят из примера AutoUpdateOptions:

```
pFIBDataSet1.SelectSQL.Text := 'SELECT * FROM EMPLOYEE';
pFIBDataSet1.AutoUpdateOptions.AutoReWriteSqls := True;
pFIBDataSet1.AutoUpdateOptions.CanChangeSQLs   := True;
pFIBDataSet1.AutoUpdateOptions.UpdateOnlyModifiedFields := True;
pFIBDataSet1.AutoUpdateOptions.UpdateTableName := 'EMPLOYEE';
pFIBDataSet1.AutoUpdateOptions.KeyFields       := 'EMP_NO';
pFIBDataSet1.AutoUpdateOptions.GeneratorName   := 'EMP_NO_GEN';
pFIBDataSet1.AutoUpdateOptions.WhenGetGenID    := wgBeforePost;
pFIBDataSet1.Open;
```

Локальная сортировка

FIBPlus содержит методы локальной сортировки, которые позволяют пересортировать кэш TpFIBDataSet в любом интересующем вас порядке, а также запомнить и восстановить сортировку после переоткрытия TpFIBDataSet. Кроме того, FIBPlus поддерживает режим, при котором вновь вставленные и измененные записи будут помещаться в правильную позицию буфера в соответствии с порядком сортировки.

Для того чтобы отсортировать буфер TpFIBDataSet вызовите один из трех следующих методов.

```
procedure DoSort(Fields: array of const; Ordering: array of Boolean); virtual;
procedure DoSortEx(Fields: array of integer; Ordering: array of Boolean);
overload;
procedure DoSortEx(Fields: TStrings; Ordering: array of Boolean); overload;
```

Первый параметр - это список полей, а второй - массив направлений сортировки, True означает, что сортировка будет по возрастанию, False – по убыванию.

Так, например, следующие два примера кода выполнят одну и ту же сортировку по двум полям в порядке возрастания:

```
pFIBDataSet1.DoSort(['FIRST_NAME', 'LAST_NAME'], [True, True]);
или
pFIBDataSet1.DoSortEx([1, 2], [True, True]);
```

Текущую сортировку можно получить, используя свойства

```
function SortFieldsCount: integer;
```

возвращает количество полей сортировки

```
function SortFieldInfo(OrderIndex:integer): TSortFieldInfo –
```

возвращает информацию о сортировке в позиции OrderIndex.

```
function SortedFields:string
```

возвращает строку с полями сортировки, перечисленными через ';'

```
TSortFieldInfo = record
  FieldName: string;          //имя поля
  InDataSetIndex: Integer;    //входит ли в индекс
  InOrderIndex: Integer;      //содержится ...
  Asc: Boolean;               //Истина если по возрастанию
  NullsFirst: Boolean;        //если Null значение перед остальными
end;
```

Для того, чтобы записи при вставке и редактировании попадали в правильную позицию буфера, установите свойство poKeepSorting в True. Если вы хотите, чтобы TpFIBDataSet автоматически использовал тот же порядок локальной сортировки, который указан в запросе в выражении ORDER BY, используйте опцию psGetOrderInfo.

Для того чтобы не терять порядок сортировки при переоткрытии TpFIBDataSet, установите свойство psPersistentSorting. Но будьте осторожны, при больших выборках это приведет к тому, что в буфер в памяти будут получены все записи с сервера, и только потом весь буфер будет отсортирован.

Сортировка национальных символов

На правильную сортировку национальных символов, как правило, влияет два параметра - набор символов (CHARSET) и порядок сортировки (COLLATION). И даже при правильно выставленных этих параметрах на уровне БД и запросов, вы можете обнаружить, что

TpFIBDataSet сортирует их неправильно. Дело в том, что при локальной сортировке по умолчанию используется простой метод сортировки без сравнения национальных кодировок. Так, как если бы был указан неопределенный набор символов (NONE).

В компоненте TpFIBDataSet существует возможность сортировки в соответствии с национальной кодировкой. Установить альтернативный метод сравнения строковых полей можно при помощи свойства OnCompareFieldValues. Существуют три готовых метода для Ansi- сортировки:

```
protected function CompareFieldValues(Field:TField;const S1,S2:variant):integer;
virtual;

public function AnsiCompareString(Field:TField;const val1, val2: variant):
Integer;

public function StdAnsiCompareString(Field:TField;const S1, S2: variant):
Integer;
```

AnsiCompareString будет учитывать регистр символов, а StdAnsiCompareString будет, наоборот, игнорировать регистр.

```
pFIBDataSet1.OnCompareFieldValues := pFIBDataSet1.AnsiCompareString;
pFIBDataSet1.OnCompareFieldValues := pFIBDataSet1.StdAnsiCompareString;
```

По умолчанию для CompareFieldValues вы можете установить один из стандартных методов, либо написать свой, если ни один из готовых вам не подходит.

Локальная фильтрация

TpFIBDataSet в отличии от IBX полностью поддерживает локальную фильтрацию. Поддерживается свойство Filter и Filtered, а также событие OnFilterRecord.

В таблице X перечислены дополнительные операторы, которые могут быть использованы в строке Filter.

Операция	***Описание***
<	Less than.
>	Greater than.
>=	Greater than or equal.
<=	Less than or equal.
=	Equal to
<>	Not equal to
AND	Logical AND
NOT	Logical NOT
OR	Logical OR
IS NULL	Tests that a field value is null
IS NOT NULL	Tests that a field value is not null
+	Adds numbers, concatenates strings, adds number to date/time values
–	Subtracts numbers, subtracts dates, or subtracts a number from a date
*	Multiplies two numbers
/	Divides two numbers

Операция	***Описание***
Upper	Upper-cases a string
Lower	Lower-cases a string
Substring	Returns the substring starting at a specified position
Trim	Trims spaces or a specified character from front and back of a string
TrimLeft	Trims spaces or a specified character from front of a string
TrimRight	Trims spaces or a specified character from back of a string
Year	Returns the year from a date/time value
Month	Returns the month from a date/time value
Day	Returns the day from a date/time value
Hour	Returns the hour from a time value
Minute	Returns the minute from a time value
Second	Returns the seconds from a time value
GetDate	Returns the current date
Date	Returns the date part of a date/time value
Time	Returns the time part of a date/time value
Like	Provides pattern matching in string comparisons
In	Tests for set inclusion
*	Wildcard for partial comparisons.

При установленном и неактивном свойстве Filter, можно воспользоваться следующими методами для перемещения по записям, которые удовлетворяют условию фильтрации:

FindFirst

FindLast

FindNext

FindPrior

На больших объемах данных рекомендуется использовать серверную фильтрацию. Как уже было сказано выше, ее можно осуществить при помощи механизма макросов и условий.

Для ознакомления же с локальной фильтрацией рекомендуется изучить пример LocalFiltering.

Важно: для того, чтобы получить количество записей, которые попадают под условие фильтрации, используйте функцию VisibleRecordCount вместо RecordCount.

Поиск данных

После фильтрации логично перейти к поиску данных. TpFIBDataSet поддерживает методы Locate, LocateNext, LocatePrior, описание которых можно получить в стандартной справке Delphi/C++Builder.

В дополнение к этим методам добавлены аналогичные методы, которые учитывают специфику FIBPlus и имеют некоторые преимущества.

```
function ExtLocate(const KeyFields: String; const KeyValues: Variant; Options:
TExtLocateOptions): Boolean;
```

```
function ExtLocateNext(const KeyFields: String; const KeyValues: Variant;
Options: TExtLocateOptions): Boolean;

function ExtLocatePrior(const KeyFields: String; const KeyValues: Variant;
Options: TExtLocateOptions): Boolean;
```

TExtLocateOptions = (eloCaseInsensitive, eloPartialKey, eloWildCards, eloInSortedDS, eloNearest, eloInFetchedRecords)

eloCaseInsensitive игнорировать регистр ;
eloPartialKey частичное совпадение
eloWildCards поиск по маске (подобно оператору LIKE);
eloInSortedDS поиск в отсортированном наборе данных (влияет на скорость);
eloNearest только в комбинации с eloInSortedDS. Позиционируется на место где "должно было быть";
eloInFetchedRecords поиск только в уже полученных записях.

Главные - подчиненные наборы данных

В дополнение к стандартному для TDataSet механизму Master-Detail, FIBPlus имеет дополнительную группу опций – DetailCondition, описанную следующим образом:

```
TDetailCondition=(dcForceOpen,dcIgnoreMasterClose,dcForceMasterRefresh,
dcWaitEndMasterScroll);
TDetailConditions= set of TDetailCondition;
```

dcForceOpen если включена эта опция, то детальные TpFIBDatSet будет открываться автоматически при открытии мастера;
dcIgnoreMasterClose если включена эта опция, то детальный TpFIBDataSet не будет закрываться в случае закрытия мастера;
dcForceMasterRefresh если включена эта опция, то при обновлении детального TpFIBDataSet будет производиться обновление текущей записи мастера;
dcWaitEndMasterScroll если включена эта опция, то TpFIBDataSet будет «ждать» некоторое время при прокрутке мастера, и лишь потом переоткрывать деталь. Опция позволяет избежать лишней работы при простой навигации по мастеру.

Пессимистическая блокировка

Стандартное поведение при обновлении записей серверами семейства InterBase заключается в оптимистической блокировке. Если одна запись одновременно редактируется двумя или более пользователями, то последнее обновление запишется в БД, без учета факта других модификаций с момента начала редактирования записи.

Как правило, если нужна пессимистическая блокировка, разработчики приложений для InterBase/Firebird используют так называемый «холостой update». Это означает, что перед редактированием записи выполняется холостое обновление записи, например, по первичному ключу:

update customer set cust_no = cust_no where cust_no = :cust_no

После этого с сервера автоматически запрашивается последняя актуальная версия записи. Такое поведение гарантирует, что запись нельзя будет обновить из другой транзакции, пока не завершится транзакция, выполнившая холостой update.

FIBPlus автоматизирует это поведение - вам достаточно включить опцию psProtectedEdit, либо вы можете использовать метод TpFIBDataSet.LockRecord.

Демонстрационный пример ProtectedEditing демонстрирует работу этой опции.

Работа в режиме ограниченного кэша

Режим ограниченного кэша впервые был предложен в компонентах gb_Datasets Сергея Спирина. Начиная с версии FIBPlus 6.0, аналогичная функциональность была реализована непосредственно в коде FIBPlus. Особенность данного режима состоит в том, что при навигации по TpFIBDataSet не происходит полное скачивание всех записей, которые возвращаются запросом. Фактически, реализуется имитация непосредственного доступа к произвольным записям за счет выполнения дополнительных запросов. Технология накладывает ряд требований на запрос. В частности, необходимым требованием является использование ORDER BY в SelectSQL. Причем, комбинация значений полей, входящих в выражение ORDER BY, должна быть уникальной. Во-вторых, для реального быстродействия желательно иметь два индекса - по возрастанию и по убыванию - для данной комбинации полей.

Технологию можно пояснить следующим простым примером. Пусть задан запрос в SelectSQL:

```
SELECT * FROM TABLE
ORDER BY FIELD
```

Мы можем получить несколько первых записей, делая последовательный fetch. Если мы хотим сразу посмотреть последние записи, то вместо полного выкачивания на клиента всех записей данного запроса, можно выполнить дополнительный запрос с обратной сортировкой:

```
SELECT * FROM TABLE
ORDER BY FIELD DESC
```

Очевидно, что последовательный fetch нескольких записей вернет нам последние записи относительно первоначального запроса. Аналогичные запросы выполняются для точного позиционирования на произвольную запись, а также на записи выше и ниже текущей:

```
SELECT * FROM TABLE
WHERE (FIELD = x)
```

```
SELECT * FROM TABLE
WHERE (FIELD < x)
ORDER BY FIELD DESC
```

```
SELECT * FROM TABLE
WHERE (FIELD > x)
ORDER BY FIELD
```

Для реализации этой технологии в TpFIBDataSet добавлено свойство:

property CacheModelOptions:TCacheModelOptions, где

```
TCacheModelOptions = class(TPersistent)
property BufferChunks: Integer ;
property CacheModelKind: TCacheModelKind ;
property PlanForDescSQLs: string ;
end;
```

BufferChunks в данном случае заменяет существующее свойство BufferChunks у TpFIBDataSet. Тип TCacheModelKind может принимать значение cmkStandard для использования стандартной работы локального буфера, и cmkLimitedBufferSize для использования новой технологии ограниченного локального буфера. Размер буфера - это

количество записей, указанных в свойстве BufferChunks. Свойство PlanForDescSQLs позволяет указать отдельный план для запросов с обратной сортировкой.

Необходимо отметить, что при использовании технологии ограниченного локального буфера существует ряд ограничений на другую функциональность TpFIBDataSet:

a) нельзя включать режим CachedUpdate;

b) свойство RecNo будет возвращать неправильные значения;

c) локальная фильтрация не поддерживается;

d) работа с BLOB-полями в текущей версии не гарантируется;

e) в PrepareOptions необходимо включать опцию psGetOrderInfo.

Работа с внутренним кэшем набора данных

TpFIBDataSet предоставляет вам несколько специальных методов для работы с внутренним буфером, в котором хранятся записи. В общем-то, данные методы превращают TpFIBDataSet в аналог TClientDataSet, ориентированный на InterBase. Единственное отличие состоит в том, что для работы с TpFIBDataSet должно существовать соединение с БД и в SelectSQL должен быть прописан правильный запрос. Несмотря на это ограничение, механизм является достаточно гибким для реализации множества «нестандартных» вещей. Например, следующий запрос создаст выборку из одного целочисленного поля и одного строкового:

select cast(0 as integer) some_id, cast('' as varchar(255)) some_name

from RDB$DATABASE.

Открыть такой TpFIBDataSet можно при помощи метода CacheOpen. После этого становятся работоспособны методы:

```
procedure CacheModify(aFields: array of integer; Values: array of Variant;
KindModify: byte );
procedure CacheEdit(aFields: array of integer; Values: array of Variant);
procedure CacheAppend(aFields: array of integer; Values: array of Variant);
overload;
procedure CacheAppend(Value: Variant; DoRefresh: boolean = False); overload;
procedure CacheInsert(aFields: array of integer; Values: array of Variant);
overload;
procedure CacheInsert(Value: Variant; DoRefresh: boolean = False); overload;
procedure CacheRefresh(FromDataSet: TDataSet; Kind: TCachRefreshKind ; FieldMap:
Tstrings);
procedure CacheRefreshByArrMap( FromDataSet: TDataSet; Kind: TCachRefreshKind;
const SourceFields, DestFields: array of string )
```

Так, чтобы добавить запись мы должны выполнить

pFIBDataSet1.CacheInsert([0,1],[255, 'string1'])

чтобы модифицировать

pFIBDataSet1.CacheModify([0,1],[255, 'string1'])

чтобы удалить из кеша, просто вызовите CacheDelete;

Методы CacheRefresh и CacheRefreshByArrMap позволяют обновить запись на основе данных из другого TpFIBDataSet.

Помните, что все эти операции не изменяют БД – все действия производятся в буфере

TpFIBDataSet.

Такую технику иногда можно использовать и в обычном режиме. Например, иногда возникает необходимость вставить запись при помощи какой-либо сложной хранимой процедуры, которая возвратит кодвставленной записи, и отобразить его в TpFIBDataSet. Для этого можно, в частности, вставить код и потом вызвать метод Refresh:

```
id := SomeInsertByProc;
pFIBDataSet1.CacheInsert([0], [1]);
pFIBDataSet1.Refresh;
```

Вы также можете, наоборот, удалить некоторые несуществующие записи из кеша.

Работа с блоб-полями

Достаточно часто желательно хранить в базе данных разнообразные неструктурированные данные: изображения, OLE-объекты, звук и т.д. Специально для этих целей существует специальный тип данных – BLOB.

SQL запросы при работе с BLOB-полями ничем не отличаются от запросов со стандартными типами полей, и работа с этими полями в TpFIBDataSet ничем не отличается от работы TDataSet. Единственным отличием от обычных типов данных является то, что для присвоения значения BLOB-параметру необходимо использовать потоки (специализированные потомки стандартного класса TStream). Для присвоения значения Blob-параметру TpFIBDataSet должен быть в состоянии редактирования (dsEdit).

Для работы с блоб-полями используется перегруженный метод TFIBCustomDataset

```
function CreateBlobStream(Field: TField; Mode: TBlobStreamMode): TStream;
override;
```

Он создает экземпляр специального внутреннего класса TFIBDSBlobStream. Скорее всего, вам не придется использовать этот класс напрямую, он нужен только для обмена данными между BLOB-параметром и потоком. Параметр Field определяет BLOB-поле, на основе которого будет создан поток. Параметр Mode определяет режим использования.

```
type TBlobStreamMode = (bmRead, bmWrite, bmReadWrite);
```

bmRead	поток используется для чтения BLOB поля;
bmWrite	поток используется для записи BLOB поля;
bmReadWrite	поток используется для модификации BLOB поля.

Для примера приведем две процедуры для сохранения файла в BLOB поле и загрузки содержимого BLOB поля в файл.

```
procedure FileToBlob(BlobField: TField; FileName: string);
var S: TStream; FileS: TFileStream;
begin
  BlobField.DataSet.Edit;
  S := BlobField.DataSet.CreateBlobStream(BlobField, bmReadWrite);
  try
    FileS := TFileStream.Create(FileName, fmOpenRead);
    S.CopyFrom(FileS, FileS.Size);
  finally
    FileS.Free;
    S.Free;
    BlobField.DataSet.Post;
  end;
end;
```

```
procedure BlobToFile(BlobField: TField; FileName: string);
var S: TStream;
    FileS: TFileStream;
begin
  if BlobField.IsNull then Exit;
  S := BlobField.DataSet.CreateBlobStream(BlobField, bmRead);
  try
    if FileExists(FileName)
    then FileS := TFileStream.Create(FileName, fmOpenWrite)
    else FileS := TFileStream.Create(FileName, fmCreate);
    FileS.CopyFrom(S, S.Size);
  finally
    S.Free;
    FileS.Free;
  end;
end;
```

Если вы используете TpFIBQuery для работы с BLOB-полями, то общий принцип остается тем же – необходимо использовать потоки, однако, в отличие от TpFIBDataSet, вам не потребуется создавать какие-то специальные потоки. TFIBSQLDA имеет встроенные методы SaveToStream и LoadFromStream. Т.е, для TpFIBQuery вам нужно было бы написать просто

pFIBQuery1.FN('BLOB_FIELD').SaveToStream(FileS);

Использование уникальных типов полей FIBPlus

FIBPlus имеет несколько уникальных полей. Это `TFIBLargeIntField`, `TFIBWideStringField`, `TFIBBooleanField`, `TFIBGuidField`.

`TFIBLargeIntField`

представляет собой поле типа BIGINT в Interbase/ Firebird.

`TFIBWideStringField`

Предназначено для строковых полей в кодировке UNICODE_FSS, стандартное TWideString практически нельзя использовать из-за ошибок VCL.

`TFIBBooleanField`

Эмулирует логическое поле. Стандартного логического поля в Interbase/Firebird нет, но с FIBPlus вы можете его довольно просто эмулировать. Для этого вы должны создать домен (INTEGER или SMALLINT), в имени которого будет содержаться подстрока BOOLEAN. В PrepareOptions TpFIBDataSet должно быть включено свойство psUseBooleanFlields. Тогда при создании объектов полей FIBPlus будет проверять имя домена, и если там содержится BOOLEAN, то для таких полей будут создаваться экземпляры TFIBBooleanField.

```
CREATE DOMAIN FIB$BOOLEAN AS SMALLINT
DEFAULT 1 NOT NULL CHECK (VALUE IN (0,1));
```

`TFIBGuidField`

Работает аналогично схеме работы TFIBBooleanField – т.е. поле должно быть объявлено через домен в имени которого должно присутсвовать GUID. Должна быть включена опция psUseGuidField. Пример объявления домена приведен ниже.

```
CREATE DOMAIN FIB$GUID AS CHAR(16) CHARACTER SET OCTETS;
```

Если AutoGenerateValue у поля выставлено в True, то при вставке значение поля будет сформировано автоматически.

Работа с полями-массивами

InterBase с самых ранних версий позволял описывать в таблицах многомерные поля массивы, делая хранение специализированных данных максимально удобным. Array-поля - это расширение InterBase, которое стандартом SQL не поддерживается, и работа с такими полями на уровне SQL-запросов крайне затруднена. Фактически, вы можете использовать массивы только поэлементно и только в операциях чтения. Чтобы изменить значения array-поля, необходимо использовать специальные команды InterBase API. FIBPlus позволяет обойтись без подобных сложностей, взяв на себя всю рутину, связанную с array-полями.

Пример работы с array-полями можно найти в демонстрационной базе EMPLOYEE из поставки сервера. Поле LANGUAGE_REQ таблицы JOB является массивом (LANGUAGE_REQ VARCHAR(15) [1:5])

Первый способ дает возможность редактировать array-поле при помощи специальных методов TpFIBDataSet ArrayFieldValue и SetArrayValue, а также методов GetArrayValues, SetArrayValue и AsQuad класса TFIBXSQLDA. Методы позволяют работать с таким полем, как с единой структурой. TFIBXSQLDA.GetArrayElement позволяет получить значение элемента массива по индексу.

Методы TpFIBDataSet.ArrayFieldValue и TFIBXSQLDA.GetArrayValues позволяют получить вариантный массив из значений поля-массива. Например, чтобы получить отдельные элементы, можно использовать следующий код:

```
var v: Variant;
with ArrayDataSet do begin
  v := ArrayFieldValue(FieldByName('LANGUAGE_REQ'));
  Edit1.Text := VarToStr(v[1]);
end;
```

Метод TpFIBDataSet.SetArrayValue и TFIBXSQLDA.SetArrayValue позволяют задать все элементы поля в виде массива вариантов:

```
with ArrayDataSet do
  SetArrayValue(FBN('LANGUAGE_REQ'), VarArrayOf([Edit1.Text, ...]));
```

В новых версиях FIBPlus это можно сделать еще проще, присваивая вариантный массив напрямую:

```
  FBN('LANGUAGE_REQ').Value := VarArrayOf([Edit1.Text, ...]);
```

Наиболее удачным местом для заполнения поля является событие BeforePost. В случае неудачной операции Update или Insert необходимо восстанавливать внутренний идентификатор массива у редактируемой записи. Для этого достаточно обновить текущую запись, вызвав метод Refresh. Это правило диктуется функциями InterBase API, и мы вынуждены его придерживаться. Таким образом, обработчик ошибок является важным моментом при работе с полями-массивами. Его можно поместить в OnPostError.

```
procedure ArrayDataSetPostError(DataSet: TDataSet; E: EDatabaseError; var
Action: TDataAction);
begin
  Action := daAbort;
  MessageDlg('Error!', mtError, [mbOk], 0);
  ArrayDataSet.Refresh;
end;
```

Функция TFIBXSQLDA.AsQuad для полей массивов и блоб-полей содержит BLOB_ID

данного поля.

Работу с полями-массивами демонстрирует примеры ArrayFields1 и ArrayFields2. В первом из них для отображения поля используется техника извлечения массива из поля, во втором - прямая выборка в SelectSQL. И в том, и в другом случае используется одинаковая техника записи поля в БД свернутым полем-массивом.

Релизы Firebird 1.5.X содержат ошибку, из-за которой для строковых полей возвращается неправильная длина. Поэтому вы можете увидеть странный последний символ «|» в строковых элементах массива.

Использование контейнеров TDataSetsContainer

Компонент TDataSetContainer позволяет централизованно обрабатывать события от разных компонентов TpFIBDataSet, а также посылать им сообщения, при получении которых они также могут производить какие-то дополнительные действия. Например, вы могли бы перед открытием всех TpFIBDataSet стандартным образом настраивать параметры отображение полей, сохранять и восстанавливать сортировку и много многое другое. Казалось бы, что подобного результат можно достичь, попросту присвоив один и тот же обработчик нескольким экземплярам TpFIBDataSet. Но удобство использования TDataSetContainer состоит, в частности, в том, что его, а, следовательно, и обработчики, можно размещать на отдельном TDataModule, чтобы вынести общий код приложения за пределы визуальных форм.

Кроме того, он может быть использован для задания общей функции локальной сортировки для присоединенных к нему TpFIBDataSet.

```
TKindDataSetEvent = (deBeforeOpen, deAfterOpen, deBeforeClose,
  deAfterClose, deBeforeInsert, deAfterInsert, deBeforeEdit, deAfterEdit,
  deBeforePost, deAfterPost, deBeforeCancel, deAfterCancel, deBeforeDelete,
  deAfterDelete, deBeforeScroll, deAfterScroll, deOnNewRecord, deOnCalcFields,
  deBeforeRefresh, deAfterRefresh)
```

Начиная с версии FIBPlus 6.4.2, контейнер может быть глобальным для всех экземпляров TpFIBDataSet. Для этого достаточно установить свойство IsGlobal в Истину.

Существует возможность строить целые цепочки контейнеров. Для этого используется свойство MasterContainer. Таким образом, можно дополнять поведение контейнеров путем «наследования» поведения MasterContainer.

Дополнительные действия при модификации данных TpFIBUpdateObject

В дополнение к стандартным модифицирующим запросам TpFIBDataSet можно задавать сколь угодно много дополнительных действий. Это осуществляется при помощи объектов TpFIBUpdateObject. Это объект наследник TpFIBQuery, своеобразный клиентский триггер, который позволяет выполнить дополнительные действия для TpFIBDataSet до или после вставки, модификации и удалении. Наследуемые свойства и методы TpFIBQuery читайте в соответствующем разделе Приложения.

Например, у вас в программе есть связка мастер-деталь – таблицы master(id, name) и detail(id, name, master_id) и при удалении записей из таблицы master вы хотели бы, чтобы автоматически удалялись зависимые данные в таблице detail. Пример немного надуманный, но использование компонента демонстрирует. Итак, чтобы реализовать необходимое поведение добавим pFIBUpdateObject. Заполиним его свойство SQL 'delete from deatail where

master_id = :id', установим свойство DataSet в датасет для таблицы master, установим KindUpdate в kuDelete и наконец скажем что выполнять его нужно перед оператором основного DataSet – ExecuteOrder oeBeforeDefault. Вот и все – теперь перед удалением записи из MasterDataSet будет производитсся оператор из ассоициированного pFIBUpdateObject.

Работа в режиме разделенных транзакций

Как было сказано в разделе «Работа с транзакциями» нужно стремиться делать обновляющие транзакции как можно короче. TpFIBDataSet имеет уникальную возможность работать сразу в контексте двух транзакций: Transaction и UpdateTransaction. Мы рекомендуем этот способ, как наиболее правильный для работы с InterBase/Firebird. Запускается одна длинная транзакция, в которой данные только читаются, и другая короткая транзакция, в контексте которой выполняются все модифицирующие запросы.

При этом читающая транзакция (Transaction), как правило, ReadCommited и только для чтения (чтобы не удерживать версии записей). Рекомендуемые параметры для нее: read, nowait, rec_version, read_committed. Пишущая транзакция (UpdateTransaction) короткая и конкурентная, рекомендуемые параметры write, nowait, concurency.

В этом случае, при использовании свойства AutoCommit = True, каждое изменение в TpFIBDataSet будет записано в БД немедленно и станет доступным для других пользователей.

Тем не менее, несмотря на рекомендацию, использования режима разделенных транзакций и AutoCommit необходимо использовать обдуманно, особенно в связках мастер-деталь. Вы должны хорошо представлять себе, в какой момент запускаются и закрываются транзакции в вашем приложении во избежание «непонятных» ситуаций.

Пакетная обработка

У компонента TpFIBDataSet есть несколько методов для выполнения пакетных операций. Это методы BatchRecordToQuery и BatchAllRecordToQuery, которые выполняют SQL-запрос предварительно настроенный в TpFIBQuery, передающийся как параметр.

Использование этих методов показано в демонстрационном примере DatasetBatching.

Централизованная обработка ошибок – TpFIBErrorHandler

FIBPlus позволяет централизованно обрабатывать ошибки и исключительные, которые возникают при работе «своих» компонентов. Для этого существует компонент TpFIBErrorHandler с единственным событием OnFIBErrorEvent:

```
TOnFIBErrorEvent = procedure(Sender: TObject; ErrorValue: EFIBError;
  KindIBError: TKindIBError; var DoRaise: boolean) of object;
где
TKindIBError = (keNoError, keException, keForeignKey, keLostConnect,
    keSecurity, keCheck, keUniqueViolation, keOther);

EFIBError = class(EDatabaseError)
  //..
  property SQLCode : Long read FSQLCode ;
  property IBErrorCode: Long read FIBErrorCode ;
  property SQLMessage : string read FSQLMessage;
  property IBMessage : string read FIBMessage;
end;
```

Опция DoRaise управляет дальнейшим поведением библиотеки после выполнения обработчика, то есть, указывает, будет ли генерироваться стандартное исключение или нет.

Опция может быть использована для стандартной обработки различных типов ошибок перечисленных в KindIBError.

Начиная с версии FIBPlus 6.4.2, в компонент были добавлены новые свойства, которые позволяют корректно работать с локализированными сообщениями сервера. Вы должны указать такие строковые свойства как Index, Constraint, Exception и At.

Получение событий TFIBSibEventAlerter

Для получения событий БД используется компонент TFIBSibEventAlerter. Необходимо установить свойство Database, чтобы показать, события какого соединения будут отслеживаться; указать в Events имена отслеживаемых событий; и задать свойство Active равным True, чтобы активировать компонент.

При получении события, на которое подписан компонент, будет выполняться обработчик OnEventAlert, объявленный как:

```
procedure (Sender: TObject; EventName: String; EventCount: Integer);
```

где, EventName – имя произошедшего события, EventCount их количество.

Помните, что события будут поступать только в момент подтверждения транзакции, в контексте которой оно наступило. Поэтому к моменту срабатывания ObEventAlert в приложении, могло произойти уже несколько событий.

Пример использование механизма событий смотрите в демонстрационном примере Events

Отладка приложений FIBPlus

FIBPlus предоставляет программисту мощные механизмы контроля и отладки SQL в приложениях. Для этого используются SQL-монитор и возможность автоматической сборки статистики SQL во время выполнения приложения.

Мониторинг SQL—запросов

За мониторинг SQL-запросов отвечает компонент TFIBSQLMonitor. Для его использования достаточно настроить тип интересующей вас информации в свойстве TraceFlags и написать обработчик события OnSQL. Эта возможность показана в демонстрационном примере SQLMonitor.

Регистрация выполняемых запросов

Компонент TFIBSQLLogger позволяет вести лог выполнения SQL запросов, а также вести статистику их выполнения.

Свойства:

property ActiveStatistics:boolean - включен ли сбор статистики.

property ActiveLogging:boolean - включено ли ведения лога

property LogFileName:string - имя файла, куда пишется лог

property StatisticsParams :TFIBStatisticsParams - какие именно

параметры включены в сбор статистики

property LogFlags: TLogFlags - какие операции логгируются

property ForceSaveLog:boolean - вести ли запись лога сразу же. Т.е. после каждого запроса немедленно идет запись в файл.

Методы

function ExistStatisticsTable:boolean; - проверяет, существует ли таблица для хранения статистики

procedure CreateStatisticsTable; - создает таблицу для хранения статистики в базе данных

procedure SaveStatisticsToDB(ForMaxExecTime:integer=0); - сохраняет накопленную статистику в таблицу. Параметр указывает, по каким запросам статистика нам интересна, а параметр указывает нижний предел времени выполнения запроса. Т.е. записывается статистика только по тем запросам, которые выполнялись дольше или столько же времени, указанного в ForMaxExecTime.

procedure SaveLog; - запись лога в файл (имеет смысл, если выключен ForceSaveLog).

Использование этого компонента показано в демонстрационном примере SQLLogger.

Репозитории FIBPlus

FIBPlus предоставляет своим пользователям три встроенных репозитория для хранения и использования настроек TpFIBDataSet, входящих в них TFields и сообщений об ошибках. Для использования репозиториев нужно позволить их использование – в компоненте TpFIBDatabase есть свойство UseRepositories (urFieldsInfo, urDataSetInfo, urErrorMessagesInfo), где вы должны указать, какие репозитории нужно использовать.

Работа со всеми типами репозиториев демонстрируется в примерах DataSetRepository, ErrorMessagesRepository, FieldsRepository.

Репозиторий наборов данных

Для использования этого репозитория в контекстном меню компонента TpFIBDataSet существует два пункта «Save to DataSets Repository table» и «Choose from dataSets Repository table». Первый позволяет сохранить основные свойства TpFIBDataSet в специальную таблицу FIB$DATASETS_INFO, а второй - загрузить свойства из ранее сохраненного в репозитории компонента. Если таблицы в БД не существует, вам будет предложено ее создать. Для того чтобы свойства конкретного TpFIBDataSet можно было сохранить в БД, вы должны указать значение свойства DataSet_ID, отличное от нуля.

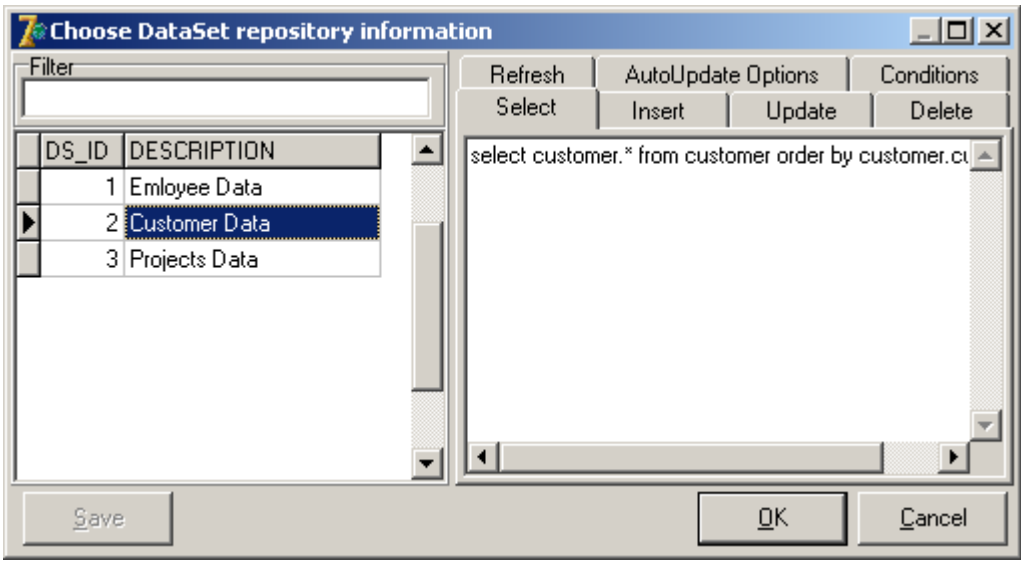

Рисунок 5. Диалог репозитория DataSets

Далее, в режиме выполнения программы достаточно просто установить DataSet_ID и свойства TpFIBDataSet будут загружены из таблицы базы данных.

Как можно увидеть из рисунка 5 сохраняются такие свойства как основные SQL-запросы, Conditions и AutoUpdateOptions.

Механизма репозиториев добавляет новый уровень гибкости в ваши приложения и позволяет изменять его поведение без перекомпиляции – достаточно реплицировать таблицы репозиториев.

Репозиторий полей

Доступ к репозиторию полей открывается через контекстное меню компонента TpFIBDatabase «Edit field information table».

Для любого поля таблицы, вида (views) и селективной процедуры вы можете задать такие свойства как DisplayLabel, DisplayWidth, Visible, DisplayFormat и EditFormat. Значение TRIGGERED позволяет пометить поля, которые будут заполнены в триггере и для которых

не требуется ввода значения в приложении, даже если поле помечено как Required (NOT NULL).

Необходимо также установить параметр psApplyRepository в TpFIBDataSet.PrepareOptions, чтобы при открытии запроса настройки TField были взяты из репозитория.

При использовании алиасов для полей в SQL запросах вы можете обнаружить, что для таких полей настройки не применяются. И это правильно, поскольку алиасов нет в физических таблицах. Тем не менее, репозиторий полей позволяет ввести настройки и для таких полей. Для этого достаточно в репозитории вместо имени таблицы написать ALIAS.

Начиная с версии FIBPlus 6.4.2 в датасет добавлени стандартное событие: TonApplyFieldRepository=**procedure**(DataSet:TDataSet;Field:TField;FieldInfo:TpFIBFieldInfo) **of object**;

который позволяет разработчику легко использовать свои собственные настройки в репозитарии полей. Например, если хочется настраивать свойство EditMask, то добавляем в таблицу репозитория поле EDIT_MASK, в приложении ставим контейнер, делаем его глобальным. В обработчике OnApplyFieldRepository пишем:

```
procedure TForm1.DataSetsContainer1ApplyFieldRepository(DataSet: TDataSet;
  Field: TField; FieldInfo: TpFIBFieldInfo);
begin
 Field.EditMask:=FieldInfo.OtherInfo.Values['EDIT_MASK'];
end;
```

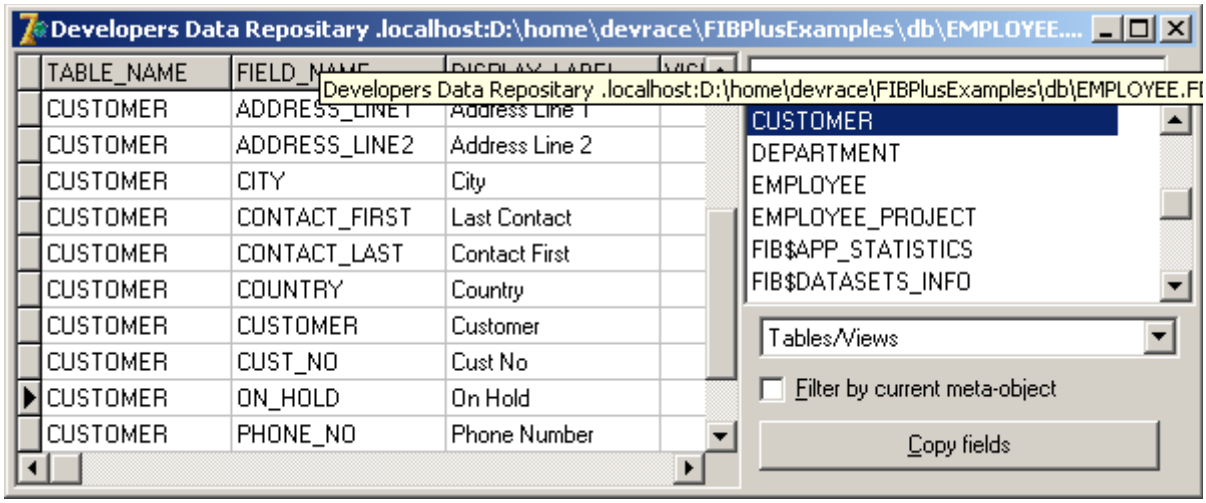

Рисунок 6. Репозиторий полей

Репозиторий ошибок

Доступ к репозиторию открывается через контекстное меню компонента TpFIBDatabase «Edit error messages table». Здесь вы можете задать свои собственные сообщения об ошибках для таких ситуаций как нарушение уникальности PK, ограничений unique, ограничений FK, ограничений checks и уникальных индексов.

Для использования репозитория необходимо поместить на какую-либо форму или модуль проекта компонент TpFIBErrorHandler, и включить использование этого репозитория. Все тексты возникающих ошибок, которые содержатся в репозитории, будут автоматически заменены вашими сообщениями.

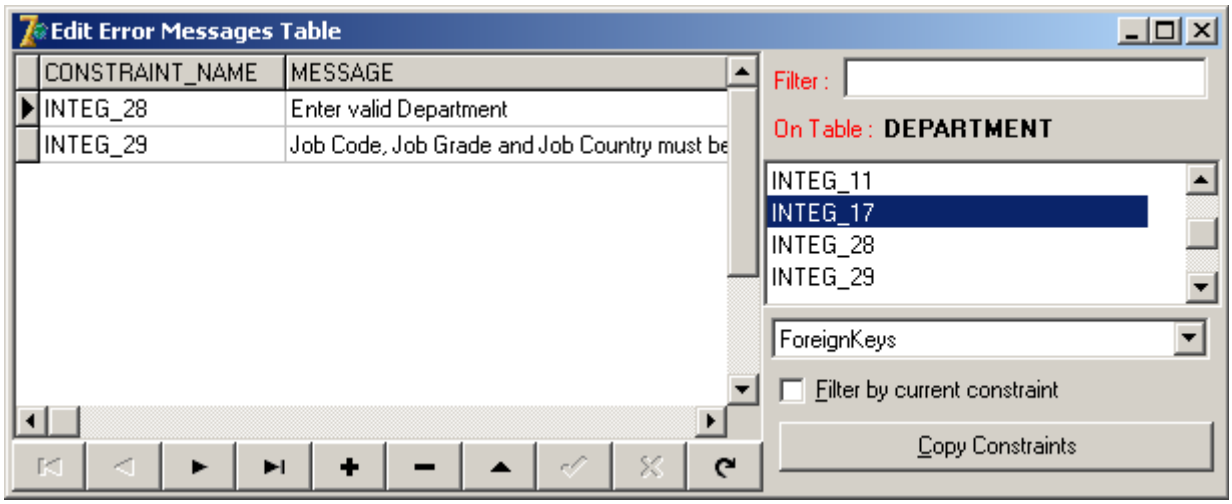

Рисунок 7. Репозиторий ошибок

Поддержка FB2.0

Несмотря на то, что на момент написания этого документа доступна только первая публичная бета-версия Firebird 2.0, FIBPlus полностью совместим с этой версией сервера.

Теперь в SelectSQL можно пользоваться конструкциями Execute block

Опция *poAskRecordCount* теперь во всех случаях работает правильно т.к. используется конструкция select count from (select – ваша оригинальная выборка).

В компонент TpFIBDatabase добавлены два метода для поддержки новых команд RDB$GET_CONTEXT и RDB$SET_CONTEXT введенных в Firebird 2:

```
function GetContextVariable (ContextSpace: TFBContextSpace; const VarName: string): Variant;
```

```
procedure SetContextVariable (ContextSpace: TFBContextSpace; const VarName, VarValue: string);
```

FIBPlus поддерживает конструкцию FB2.0 insert ... into ... returning. Теперь можно не заботиться о получении с клиента значения генератора, а оставлять его на триггере. Появилась возможность использования RDB$DB_KEY. Новые возможные техники работы с insert returning и RDB$DB_KEY показаны в примере FB2InsertReturnin.

Если у вас есть запросы соединения таблицы с самой собой:

Select * from Table1 t, Table1 t1

where ....

то только в версии Firebird 2.0 для каждого поля FIBPlus может четко «понять», откуда оно взялось: из t или t1. Эта особенность используется внутри компонентов FIBPlus для правильной генерации модифицирующих запросов.

Дополнительные возможности

Полная поддержка UNICODE_FSS

С версии 6.0 FIBPlus поддерживает правильную работу с CHARSET UNICODE_FSS. Для нормальной работы вам также понадобятся визуальные компоненты, поддерживающие юникод, например TntControls.

В связи с поддержкой юникод добавлены новые типы полей:

TFIBWideStringField = class(TWideStringField) - для полей типа VARCHAR,CHAR;
TFIBMemoField = class(TMemoField, IWideStringField) - для BLOB-полей, где
IWideStringField - интерфейс обрабатываемый визуальными компонентами TNT
(http://tnt.ccci.org/delphi_unicode_controls)

Опция psSupportUnicodeBlobs в TpFIBDataSet.PrepareOptions позволяет работать с BLOB-полями UNICODE_FSS. По умолчанию опция отключена, потому что для выяснения charset конкретного BLOB-поля необходим дополнительный запрос. Если вы не работаете с UNICODE, то запрос окажется лишним.

В TpFIBDatabase реализован метод: function IsUnicodeCharSet: Boolean, который возвращает True, если подключение использует UNICODE_FSS.

В класс FIBXSQLVAR реализовано свойство AsWideString: WideString возвращающее WideString.

Опция компиляции NO_GUI

Начиная с версии 6.0, в FIBPlus введена новая директива компиляции NO_GUI. При ее использовании библиотека не ссылается на стандартные модули, которые содержат визуальные компоненты, что позволяет получать приложения, ориентированные на системные, а не пользовательские задачи. {$DEFINE NO_GUI} в файле FIBPlus.inc.

Использование SynEdit в редакторах

Если у вас в установлен набор компонентов SynEdit, вы можете скомпилировать пакеты редакторов с использованием SynEdit. После этого в SQL-редакторе вам будет доступно синтаксическое выделение SQL операторов и инструмент CodeComplete. За использование SynEdit отвечает директива {$DEFINE USE_SYNEDIT} в файле pFIBPropEd.inc.

Важно: вы не можете изменять директивы компиляции в триальной версии FIBPlus.

Уникальное расширение FIBPlusTools

Кроме компонент библиотека FIBPlus также включает ряд дополнительных инструментов - FIBPlus Tools, которые расширяют возможности среды разработки для более удобного и эффективного использования FIBPlus в design-time.

Preferences

Пункт Preferences позволяет настроить параметры основных компонент по умолчанию. На первой странице диалога вы можете задать значения по умолчанию для свойств Options, PrepareOptions и DetailsConditions для всех компонент класса TpFIBDataSet. Вы можете задать определенные ключи для этих свойств. Например, если вы включите флажок SetRequiredFields то, когда вы положите новую компоненту TpFIBDataSet на вашу форму, ее свойство PrepareOptions будет содержать ключ pfSetRequiredFields. Наиболее важным

является тот факт, что умолчания, заданные в FIBPlus Tools Preferences, действуют во всех приложениях, которые вы будете создавать. Однако необходимо иметь в виду, что это только первоначальные умолчания. То есть, если после помещения компоненты на форму вы измените свойства, то это никак не коснется Preferences. Также изменение Preferences не коснется тех компонент, значения свойств которых уже были заданы.

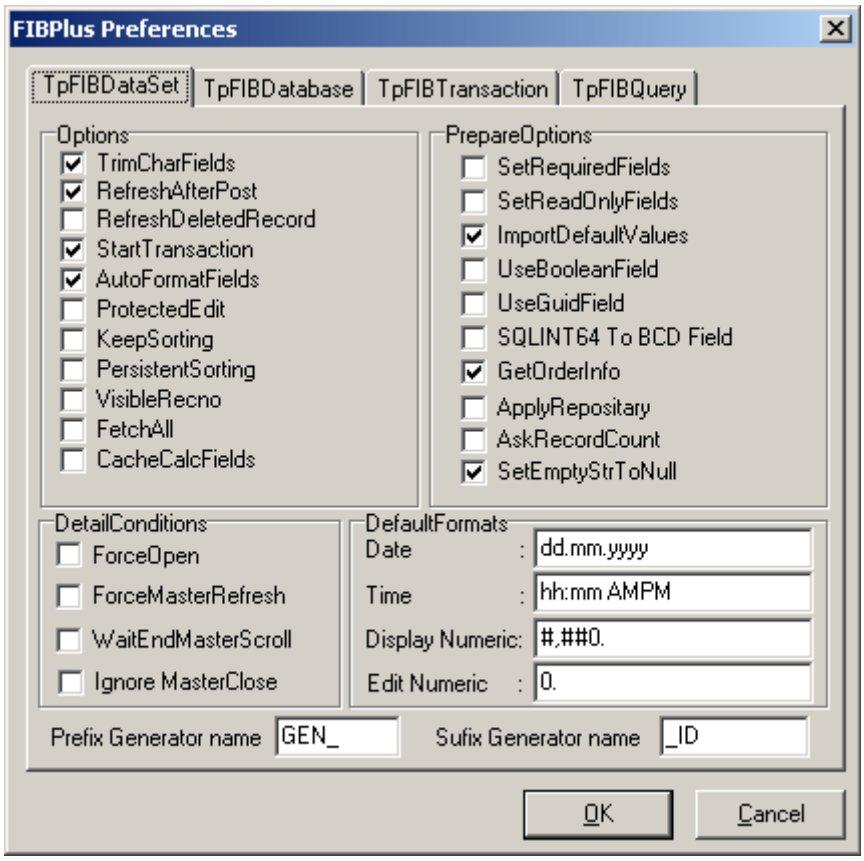

Рисунок 8. Tools Preferences.

Обратите внимание на поля Prefix Generator name и Suffix Generator name. Задав их значения, вы сможете регулировать формирование имен для названий генераторов в свойстве AutoUpdateOptions у TpFIBDataSet. Имя генератора в AutoUpdateOptions генерируется из названия таблицы (UpdateTable), префикса и суффикса.

Следующие страницы диалога позволяют настраивать ключевые свойства компонентов TpFIBDataBase, TpFIBTransaction и TpFIBQuery. В частности, если вы всегда работаете с новыми версиями InterBase, то есть, с версиями 6 и более (а также Firebird), то мы рекомендуем вам задать значение SQL Dialect на закладке TpFIBDatabase равным 3, чтобы каждый раз не переключать это свойство «вручную».

SQL Navigator

Это наиболее интересная часть FIBPlus Tools, не имеющая аналогов в других продуктах. Фактически, это инструмент централизованной обработки SQL в рамках целого приложения.

SQL Navigator позволяет разработчику получить доступ к свойствам SQL любого компонента приложения и все это в одном месте.

Кнопка «Scan all forms of active project» сканирует все формы приложения и выделяет из них те, которые содержат компоненты FIBPlus для работы с SQL: TpFIBDataSet, TpFIBQuery, TpFIBUpdateObject и TpFIBStoredProc. Нажмите в списке на любую из обнаруженных форм. Список справа будет заполнен компонентами, обнаруженными на этой

форме. Нажатие на любой из компонентов позволит нам посмотреть соответствующие свойства, в которых содержится SQL-код. Для компонентов класса TpFIBDataSet будут выведены свойства: SelectSQL, InsertSQL, UpdateSQL, DeleteSQL и RefreshSQL. Для компонент TpFIBQuery, TpFIBUpdateObject и TpFIBStoredProc будет выведено значение свойства SQL.

Вы можете изменить любое свойство напрямую из SQLNavigator, и новое значение будет сохранено. SQLNavigator позволяет делать операции с группами компонент. Для этого достаточно пометить соответствующие компоненты или даже формы.

“Save selected SQLs” сохраняет значения выделенных свойств во внешний файл.

“Check selected SQLs” проверяет корректность выделенных запросов прямо в SQLNavigator. Записанный файл с выделенными запросами можно использовать для дальнейшего анализа при помощи специализированных инструментов.

Вы можете использовать SQLNavigator для поиска текста в SQL в рамках всего проекта.

При помощи двойного нажатия на каждом найденном элементе SQLNavigator выберет компонент и свойство, чтобы разработчик мог редактировать SQL.

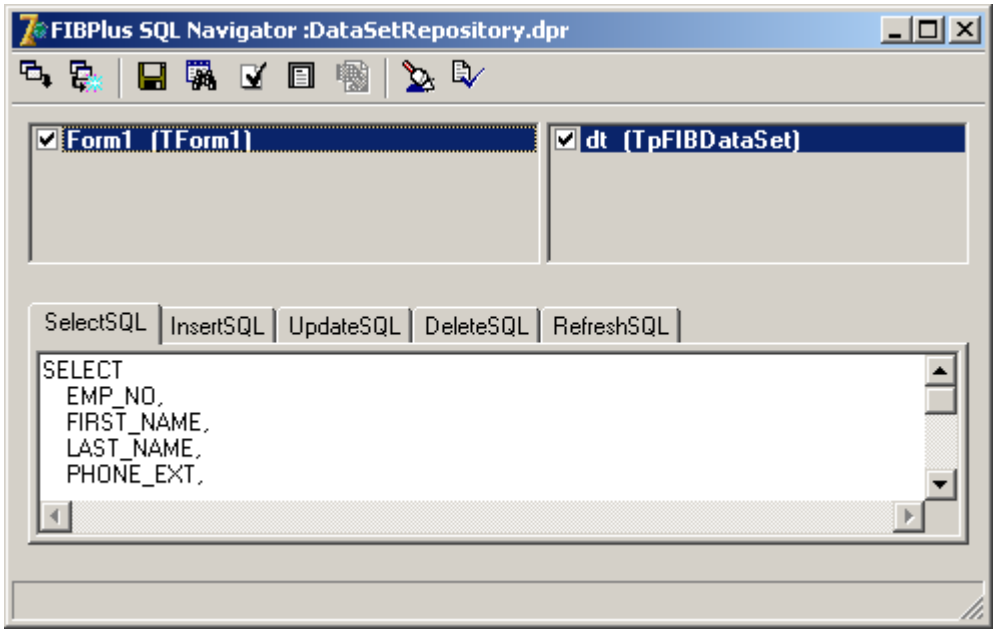

Рисунок 9. Tools SQL Navigator

Работа с сервисами

В дополнение к основной вкладке палитры компонентов FIBPlus библиотека содержит еще одну вкладку: FIBPlus Services. Собранные здесь компоненты предназначены для получения информации о сервере InterBase/Firebird, управления пользователями, тонкой настройки параметров баз данных и их обслуживания.

Подробную информацию по всем этим вопросам вы можете найти в документации к InterBase (OpGuide.pdf, разделы «Database Security», «Database Configuration and Maintenance», «Database Backup and Restorre» и «Database and Server Statistic»).

Вся работа, которая может быть выполнена при помощи сервисов, показана в демонстрационном примере Services. Кроме того, есть хороший пример в поставке Delphi/C++Builder по пути DelphiX\Demos\Db\IBX\Admin\. Несмотря на то, что он написан с использованием IBX, этот пример подойдет и для FIBPlus.

Все компоненты сервисов унаследованы от класса TpFIBCustomService и имеют ряд обязательных свойств и методов. Это, в частности, ServerName - имя сервера, к которому будет производиться подключение, Protocol - сетевой протокол подключения, UserName - имя пользователя, и Password - пароль пользователя.

В общем случае работа с сервисом выглядит следующим образом. Сначала производится установка свойств подключения (сервер, протокол, пользователь и пароль). Потом присоединение к серверу (Attach := True), выполнение необходимых действий и отсоединение (Attach := False). Данная схема представлена в коде ниже и необходима при работе с любым сервисом.

```
with TpFIBXXXService.Create(nil) do
try
  ServerName := <server_name>;
  Protocol := <net_protocol>;
  UserName := <user_name>;
  Password := <password>;
  Atach;
  try
    //выполняемая работа
  finally
    Deatach;
  end;
finally
  Free;
end;
```

Получение информации о сервере

Вы можете получить информацию о лицензиях, конфигурации сервера, количестве подключенных пользователей и базах данных, используя компонент TpFIBServerProperties. О доступных свойствах, событиях и методах подробнее смотрите в Приложении. В общем случае вам нужно соединиться с сервером, установить интересующую вас информацию и получить ее при помощи метода Fetch.

Параметры и результаты описаны в модуле IB_Services.pas:

Следующая запись содержит информацию о количестве подключений к серверу, количестве активных БД, обслуживаемых сервером, и именах этих активных баз данных:

```
  TDatabaseInfo = record
    NoOfAttachments: Integer;
    NoOfDatabases: Integer;
```

```
    DbName: Variant;
  end;
```

Следующие две записи содержат информацию о лицензиях сервера InterBase:

```
  TLicenseInfo = record
    Key: Variant;
    Id:     Variant;
    Desc:    Variant;
    LicensedUsers: Integer;
  end;

  TLicenseMaskInfo = record
    LicenseMask: Integer;
    CapabilityMask: Integer;
  end;

  TConfigFileData = record
    ConfigFileValue:Variant;
    ConfigFileKey:Variant;
  end;

  TConfigParams = record
    ConfigFileData: TConfigFileData;
    BaseLocation: string;
    LockFileLocation: string;
    MessageFileLocation: string;
    SecurityDatabaseLocation: string;
  end;
```

Следующая запись содержит информацию о версии сервера:

```
  TVersionInfo = record
    ServerVersion: String;
    ServerImplementation: string;
    ServiceVersion: Integer;
  end;

  TPropertyOption = (Database, License, LicenseMask, ConfigParameters, Version);
  TPropertyOptions = set of TPropertyOption;

  TpFIBServerProperties = class(TpFIBCustomService)
    procedure Fetch;
    procedure FetchDatabaseInfo;
    procedure FetchLicenseInfo;
    procedure FetchLicenseMaskInfo;
    procedure FetchConfigParams;
    procedure FetchVersionInfo;
    property DatabaseInfo: TDatabaseInfo
    property LicenseInfo: TLicenseInfo
    property LicenseMaskInfo: TLicenseMaskInfo
    property ConfigParams: TConfigParams
    property Options : TPropertyOptions
  end;
```

Для получения лога работы сервера используется компонент TpFIBLogService.

Многие сервисы возвращают текстовую информацию. Такая информация возвращается при помощи события OnTextNotify типа TserviceGetTextNotify. Этот тип описан в модуле IB_Services следующим образом:

```
  TServiceGetTextNotify = procedure (Sender: TObject; const Text: string) of
```

```
object;
```

TpFIBLogService использует именно такой тип оповещения. Работа с этими сервисами требует установки реакции на событие OnTextNotify старта сервиса и вычитывания информации. Выглядит работа по вычитыванию информации всегда одинаково:

```
    ServiceStart;
    while not Eof do
      GetNextLine;
```

Т.е. полный код будет выглядеть следующим образом:

```
with TpFIBXXXService.Create(nil) do
try
  ServerName := <server_name>;
  Protocol := <net_protocol>;
  UserName := <user_name>;
  Password := <password>;
  Atach;
  try
    ServiceStart;
    while not Eof do
      GetNextLine;
  finally
    Deatach;
  end;
finally
  Free;
end;
```

При работе с некоторыми другими аналогичными сервисами дополнительно требуется установка опций.

Подробности смотрите в примере Services.

Управление пользователями сервера

Работу с пользователями обеспечивает компонент TpFIBSecurityService. Он содержит методы для получения информации о пользователях, а также для добавления, модификация и удаления пользователей. Пользователю этого компонента доступна следующая информация:

Запись, предоставляющая информацию о пользователе на сервере:

```
  TUserInfo = class
  public
    UserName: string;
    FirstName: string;
    MiddleName: string;
    LastName: string;
    GroupID: Integer;
    UserID: Integer;
  end;
```

Доступные свойства и методы компонента:

```
  TpFIBSecurityService = class(TpFIBControlAndQueryService)
    procedure DisplayUsers;
    procedure DisplayUser(UserName: string);
    procedure AddUser;
    procedure DeleteUser;
```

```
    procedure ModifyUser;
    procedure ClearParams;
    property  UserInfo[Index: Integer]: TUserInfo read GetUserInfo;
    property  UserInfoCount: Integer read GetUserInfoCount;

    property SQlRole : string read FSQLRole write FSQLrole;
    property UserName : string read FUserName write FUserName;
    property FirstName : string read FFirstName write SetFirstName;
    property MiddleName : string read FMiddleName write SetMiddleName;
    property LastName : string read FLastName write SetLastName;
    property UserID : Integer read FUserID write SetUserID;
    property GroupID : Integer read FGroupID write SetGroupID;
    property Password : string read FPassword write setPassword;
  end;
```

Для того, чтобы получить информацию о пользователе сервера, используется метод DisplayUser, параметром которого является имя интересующего нас пользователя. Для того, чтобы получить информацию обо всех пользователях, используется метод DisplayUsers. После вызова одного из этих методов свойство UserInfoCount будет содержать количество пользователей, а индексное свойство UserInfo будет возвращать запись о пользователе по индексу.

Для того, чтобы добавить пользователя, вам нужно заполнить свойства UserName, FirstName, MiddleName, LasName, Password и выполнить метод AddUser. Аналогично AddUser работают DeleteUser и ModifyUser; минимальным необходимым параметров для них будет имя пользователя UserName.

Обратите внимание на свойство Password – оно не возвращается методами DisplayUser и DisplayUsers. Также свойство SQLRole не может быть использовано для ассоциирования роли с пользователем. Используйте для этих операций выполнение запросов через TpFIBQuery - GRANT/REVOKE.

Все аспекты работы с пользователями сервера представлены в демонстрационном примере Services

Конфигурирование базы данных

TpFIBConfigService позволяет осуществлять настройку таких параметров, как:

- интервал в транзакциях до автоматической сборки мусора (Sweep Interval);
- установка режима записи изменений на диск (Async Mode);
- установка размера страницы БД;
- установка резервируемого пространства для БД;
- установка режима доступа только для чтения;
- активация и деактивация тени

А также позволяет выполнять такие действия, как:

- остановка базы данных (shutdown)
- старт базы данных (online).

Прежде всего, для работы с сервисом вам нужно установить свойство

```
  property DatabaseName: string
```

Путь к базе данных на сервере

Чтобы установить интервал в транзакциях для автоматической сборки мусора, используется метод `SetSweepInterval`, единственным параметром которого нужно выставить интересующий вас интервал (по умолчанию он равен 20000)

```
procedure SetSweepInterval (Value: Integer);
```

Для установки диалекта базы данных используется метод SetDBSqlDialect, параметром которого есть диалект БД. Поддерживается всего три параметра 1, 2, 3.

```
procedure SetDBSqlDialect (Value: Integer);
```

Для установки PageBuffers используется метод SetPageBuffers, параметром которого есть интересующий размер буфера.

```
procedure SetPageBuffers (Value: Integer);
```

Для активации тени используется метод ActivateShadow. Параметры для этого метода не требуются.

```
procedure ActivateShadow;
```

Для установки режима асинхронной записи на диск служит метод SetAsyncMode с параметром Истина, для снятия режима передайте параметром Ложь

```
procedure SetAsyncMode (Value: Boolean);
```

Для установки режима только для чтения испльзуется метод SetReadOnly. Передайте Истину, если хотите установить режим только для чтение и Ложь, если хотите снять. Режим только для чтения испльзуется для подготовки баз для распространения на носителях типа компакт-дисков.

```
procedure SetReadOnly (Value: Boolean);
```

Для установки ... служит метод SetReserveSpace

```
procedure SetReserveSpace (Value: Boolean);
```

Для остановки доступа к БД используется метод ShutdownDatabase. Возможны три стандартных варианта остановки БД – форсированное, с запретом новых транзакций и с запретом новых соединений. Все вышеперечисленные операции желательно выполнять с монопольным подключением к бд администратором сервера - SYSDBA

```
procedure ShutdownDatabase (Options: TShutdownMode; Wait: Integer);

TShutdownMode = (Forced, DenyTransaction, DenyAttachment);
```

Для возвращения БД в состояние доступное для подключения используется метод BringDatabaseOnline.

```
procedure BringDatabaseOnline;
```

Обслуживание базы данных

Компоненты TpFIBBackupService и TpFIBRestoreService открывают доступ к функциям резервирования и восстановления баз данных.

TpFIBValidationService позволяет выполнить сборку мусора, проверить базу данных на ошибки, и, в случае поломки, выполнить починку БД.

При резервировании, восстановлении и проверке баз данных большое влияние оказывают опции. Подробное описание всех этих опций содержится в документации к серверу OpGuide.pdf.

Работа с этими сервисам крайне проста и мало чем отличается от работы FIBLogService. Единственной сложностью может стать интерпретация результатов работы этих сервисов. Тут есть некоторые особенности.

Если нужно убедится, что не было ошибок при резервировании или восстановлении БД нужно смотреть лог выполнения операции. При возникновении ошибок лог операции будет содержать строку вида «GBAK: ERROR»

Если TpFIBValidationService не нашел ошибок, его лог будет состоять из одной пустой строки.

Если возникли ошибки при починке базы данных (опция Mend TpFIBValidationService), то они будут записаны в лог самого сервера.

Важно помнить, что при работе TpFIBBackupService файл бекапа базы данных создается на машине сервере – т.е. резервирование делает сервер и, соответственно, файл создается на сервере. Но резервировать можно файл с другого сервера. Тогда в строке подключения к базе данных нужно писать полный путь к БД. Выглядеть это будет примерно так:

```
with TpFIBXXXService.Create(nil) do
try
  ServerName := <local_backup_server_name>;
  Protocol := <net_protocol>;
  UserName := <user_name>;
  Password := <password>;
  DatabaseName := <remote_db_name>;
  BackupFile.Add(<local_backup_name>);
  Atach;
  try
    ServiceStart;
    while not Eof do
      GetNextLine;
  finally
    Deatach;
  end;
finally
  Free;
end;
```

Описанные сервисы могут работать как обычным сервером, так и со встроенным сервером. Единственное условие: не забудьте указать протокол Local при работе с Firebird Embedded Server.

Получение статистической информации о состоянии БД

При помощи компонента TpFIBStatisticalService можно получить важную статистическую информацию о работающей БД. Работа получения статистики ничем не отличается от работы с TpFIBLogService.