

Tutorial Using FIBPlus

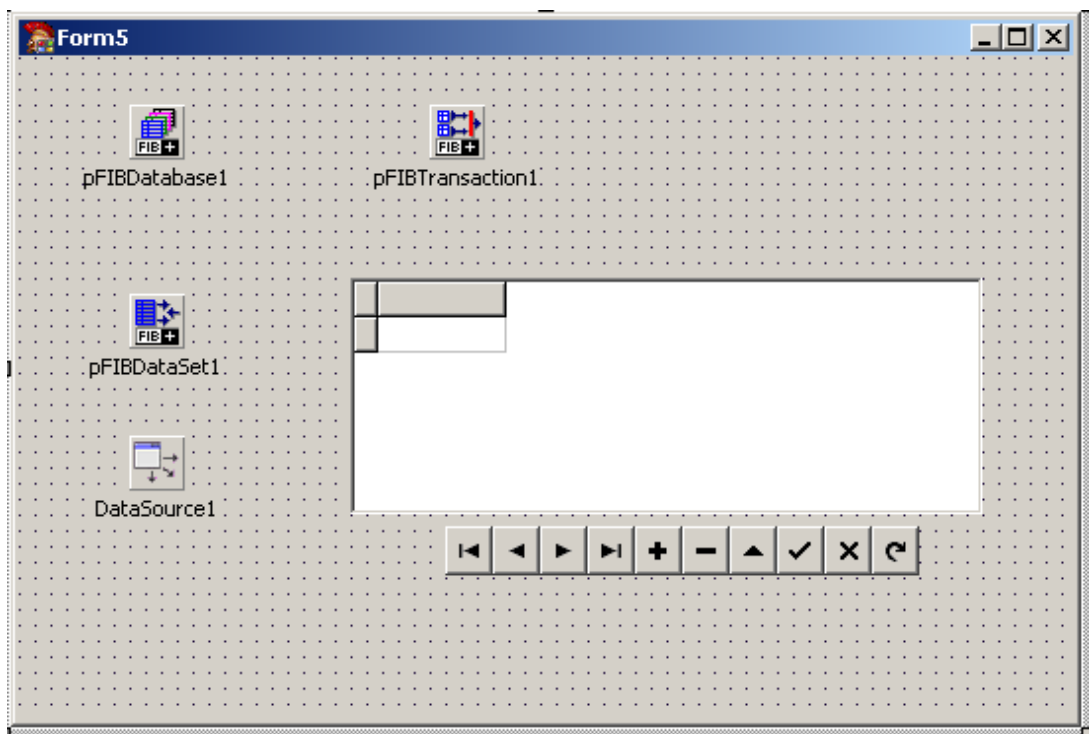
Introduction:

- This is a collection of many articles and tutorials gathered to the benefit of new programmers who will work with InterBase and/or Firebird. FIBPlus is definitely the best efficient driver for InterBase and Firebird, the driver accesses the database directly and they optimized several standard database methods to make it a lot faster than native IBX components.

- For people that came from BDE background stop using BDE it is outdated and no more suitable for today's modern programming. There is also a methodology change in the way you suppose to work with SQL database like InterBase and Firebird summarized as following.

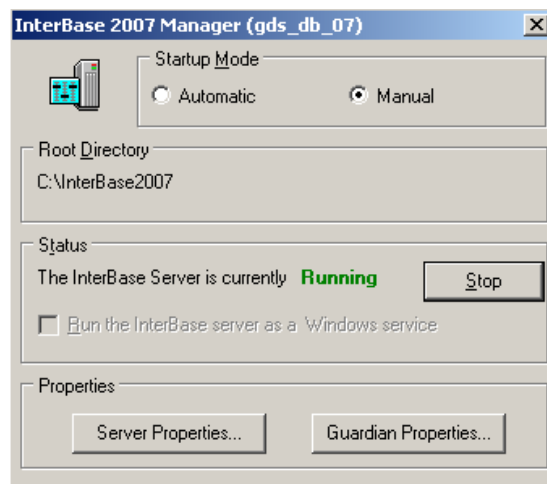
- You can set transaction parameters even in run-time.
- You can work with multi-database transactions.
- You can create "alive"-queries to several tables.
- FIBPlus based applications do not require BDE. It is easier to deploy.
- You can use InterBase SQL roles.
- Many other advantages

- In the following pages you will find steps how to use FIBPlus, first you will need to add the following components as shown in the next picture

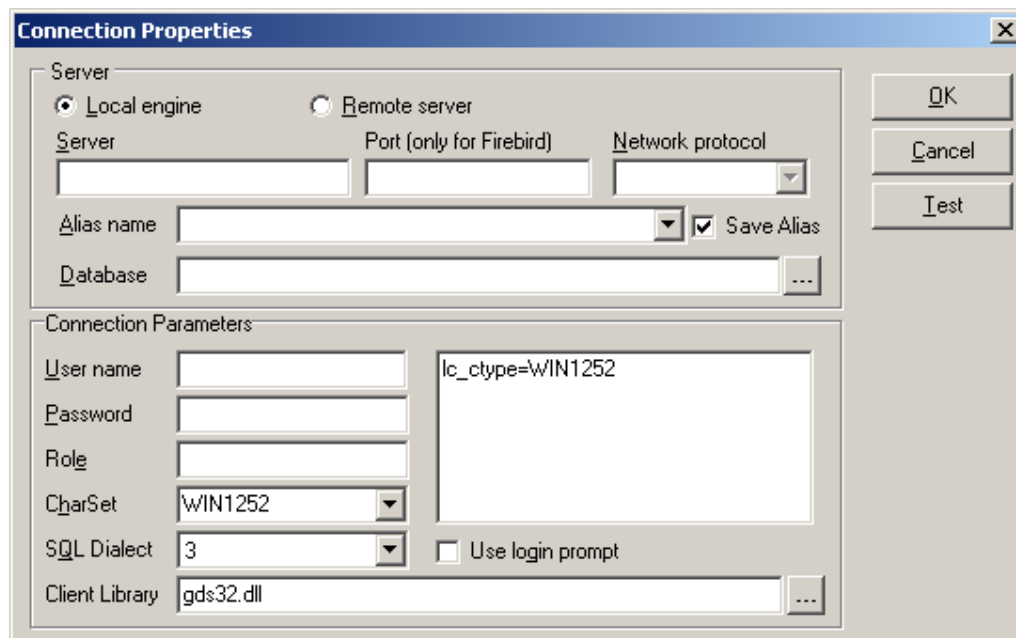


- Make sure first you are running the InterBase (or Firebird) server, you can run the server as a windows service or as an application to know exactly if the server is running open the "InterBase Server Manager" from the start menu

- you can see a dialog will open showing the status of the running server, you also choose to run the server at every windows startup or manually



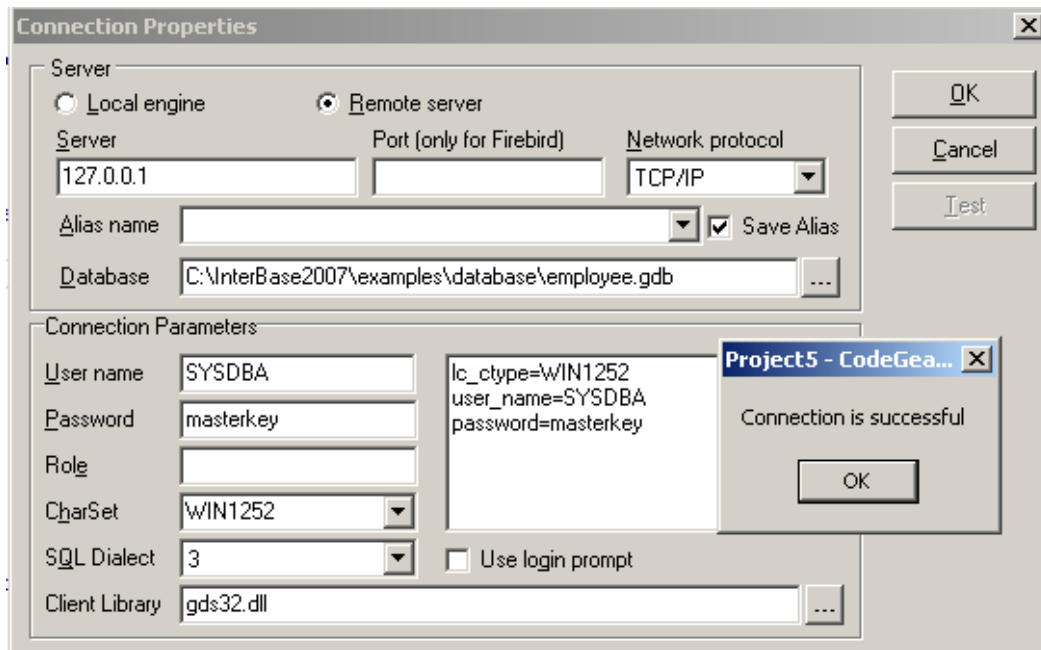
- Now we will set the basic properties to make a small database application - First the Database component, the easiest way is to right click on the component and choose "Database Editor"



- You can choose two types of connection: "Local" to connect for a database in this PC or "Remote" to connect to another PC have the database file
- For local you need to point to a database file (*.IB, *.gdb or *.fbd) you can find a ready to use database in the following path:
<InterBase installation folder>\examples\database\employee.gdb
- Default User Name and Password are : SYSDBA/masterkey.
- Leave SQL Dialect at 3 and client library to gds32.dll for InterBase and fbclient.dll for Firebird

- For remote connection , type the server IP or name (IP is faster) and choose the connection protocol TCP/IP, in the example I used IP 127.0.0.1 which is the local machine

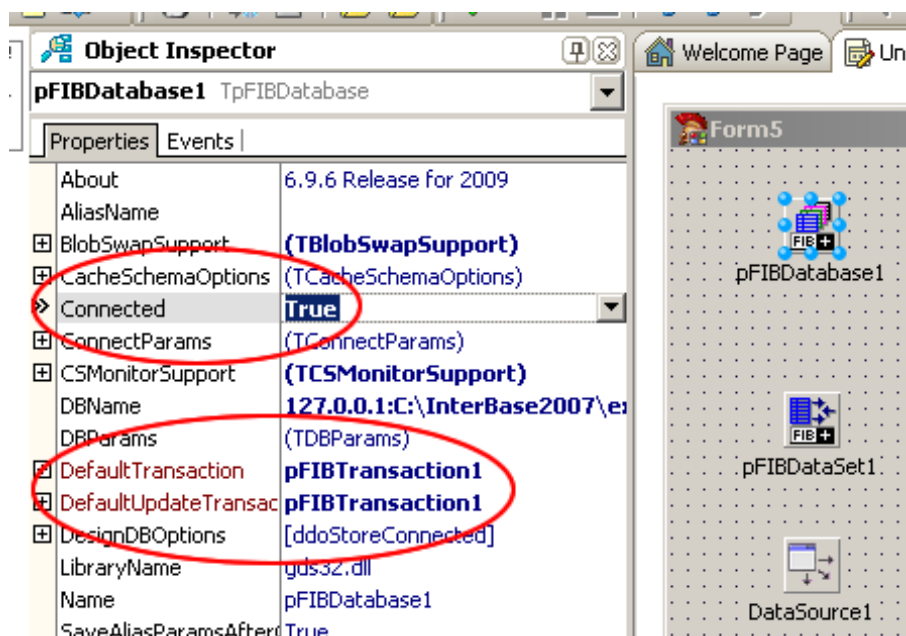
- If you press on test button now you should get a succell connection message



- Now press on Ok button, one last property we need to setup is the "defaultTransaction" property and the "defaultUpdateTranaction" property – FIB gives you ability to set different transaction for reading and writing to get more speed in case you need to read only data from a table but we will set both now to the same transaction component .

TIP:

Transaction or isolation levels is way to organize data flows (transactions) between several concurrent reading and writing, for example when different users (PCs) open the same table at the same time.

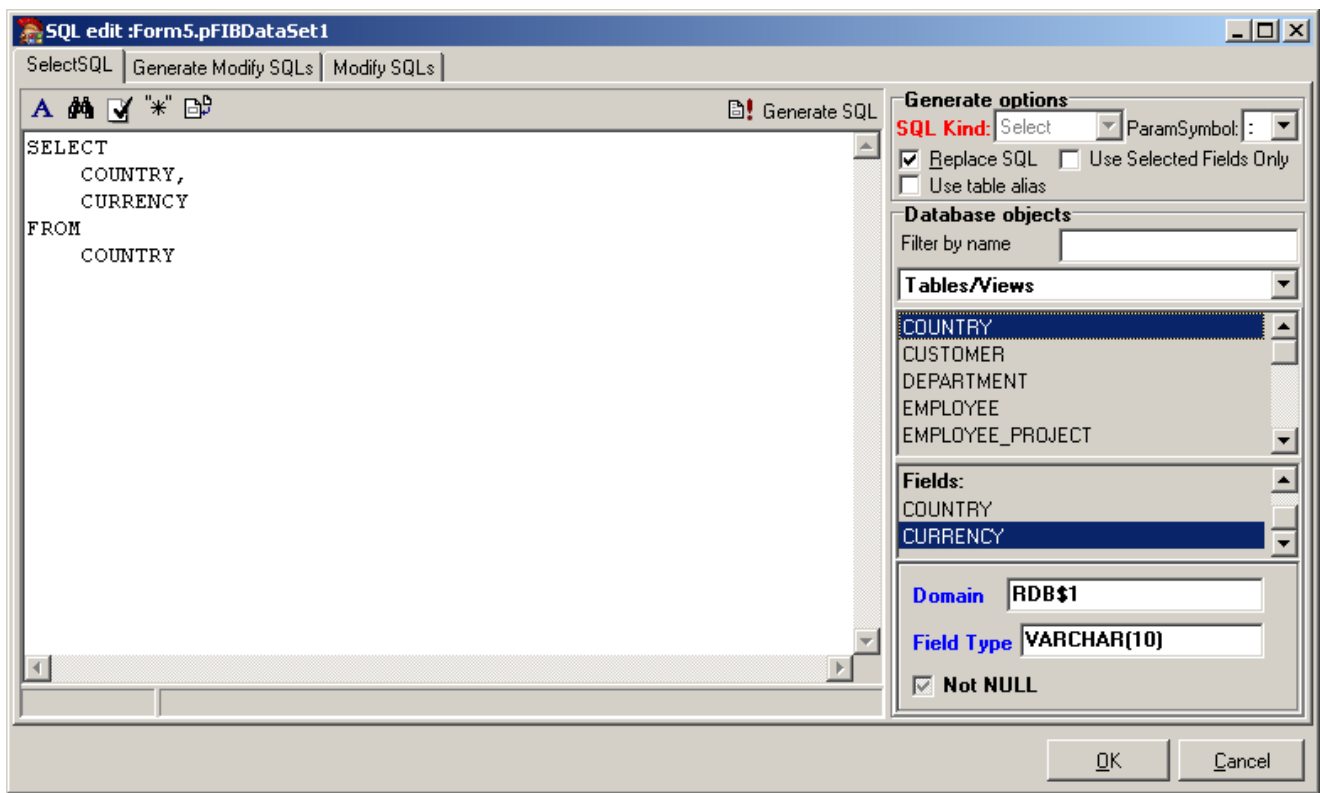


- In the transaction component you can set different transaction variables for different type of desired transactions (Isolation), I will describe at full detail later all these variables but for now leave it as it is.
- The transaction component has the active property but it is not necessary to open it manually it will be "True" when the table active property is "True"
- First Select the dataset component and make sure to point all of the "Database", "Transaction" and "UpdateTransaction" to the proper components.
- Now we will set the dataset necessary SQLs, if you remember we now deal with SQL result set and all data modifications done though SQL statement.

TIP:

If you are not familiar with InterBase SQL you should take a look at the following SQLs meanings in the SQL reference manual installed with the InterBase.

- Fortunately the dataset component has a wizard able to us to set all required SQLs quickly you can invoke this wizard by right select on the dataset and choose "SQL Generator"



- the right side has a list of tables/fields all you need to do to set the "Select" SQL is to double click on the desired table, I choose the country table.

TIP

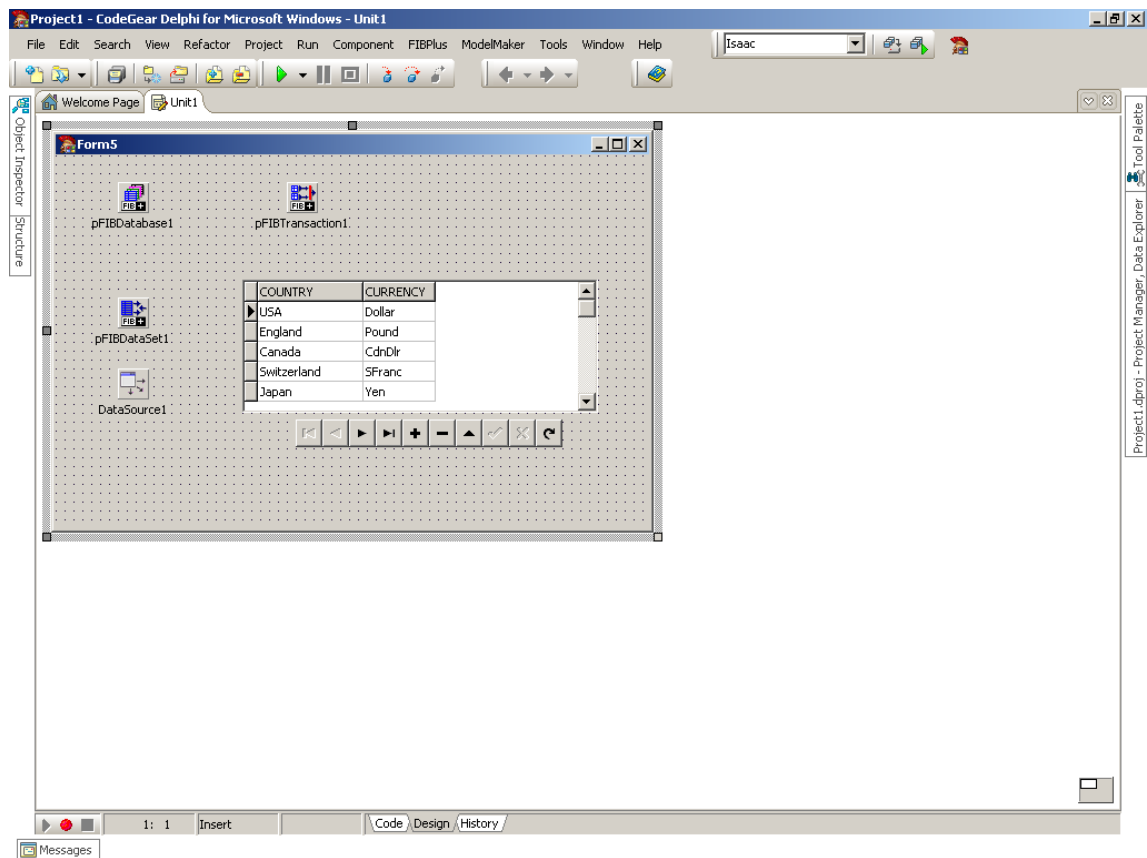
Don't use the symbol * for selecting all fields it is not the optimize way for Select.

- Now go to the "Generate modify SQLs" tab and make sure the option "Where by primary key" is selected and the option "None update primary key" is not selected so we can generate updatable SQLs based on primary key.

- Press on "get table fields" button and then on "Generate SQLs" respectively and then all necessary SQLs will be set, you can see that by selecting statement type it is also a good way to know what is the suitable SQL for Delete , Update, Select and Insert new rows.

- Make sure to press OK and now place a "Datasource" component and set the dataset property to the dataset component also drop the some data controls on the form and set the "datasource" property for each.

- You can now make the dataset active property to true to see the data populated at design time.



- Now you can simply run the application (F9) and test insert/delete/refresh SQLs.

Transactions:

By: Aleksandr Bondar

Part 1:

Transactions in multi-user environment in InterBase/Firebird (and other database servers) are often not the trivial topic. Programmers prefer to use only the isolation level READ COMMITTED regardless the task conditions.

I will consider how to use transactions and their characteristics for work with FIBPlus components.

Besides such documentation as InterBase Language Reference, Embedded SQL Guide and API Guide, and "The Firebird Book: A Reference for Database Developers" by Helen Borrie there are many articles about transactions. I won't repeat any of them but tell about their main characteristics and their influence on multi-user database environment.

I have written some test applications based on FIBPlus to experiment with transactions. Here are some results of work with the TpFIBTransaction component, checking whether transaction parameters correspond to the SQL language used to describe transaction characteristics.

Database description

I have created a new database FIBTRANSACT.FDB, which supports Firebird 1.5. There are two tables: a country reference REFCOUNTRY and a region reference REFREGION (with a list of some regions). This is a part of real database. At first you need to create domains:

```
CREATE DOMAIN DCodCtr AS CHAR(3);
CREATE DOMAIN DName30 AS VARCHAR(30) COLLATE PXW_CYRL;
CREATE DOMAIN DName60 AS VARCHAR(60) COLLATE PXW_CYRL;
CREATE DOMAIN DDescr AS BLOB SUB_TYPE 1 SEGMENT SIZE 400;
```

Table creation script (partially):

```
/** Country reference REFCOUNTRY */
CREATE TABLE REFCOUNTRY
( Name DName30, /* Short name of country */
  FullName DName60, /* Full name of country */
  CodCtr DCodCtr NOT NULL, /* Country code */
  Capital DName30, /* Capital */
  Region DName30, /* Region name */
  Description DDescr, /* Additional information */
  CONSTRAINT "Country_PRIMARY_KEY" PRIMARY KEY (CodCtr)
);
COMMIT;
/** Region reference REFREGION */
CREATE TABLE REFREGION
( CodCtr DCodCtr NOT NULL, /* Country code */
  CodReg DCodCtr NOT NULL, /* Region code */
  Center DName30, /* Name of the region centre */
  RegName DName60, /* Region name */
  Description DDescr, /* Additional information */
  CONSTRAINT "Region_PRIMARY_KEY"
  PRIMARY KEY (CodCtr, CodReg),
  CONSTRAINT "Region_FOREIGN_KEY"
  FOREIGN KEY (CodCtr) REFERENCES REFCOUNTRY (CodCtr)
  ON DELETE CASCADE
```

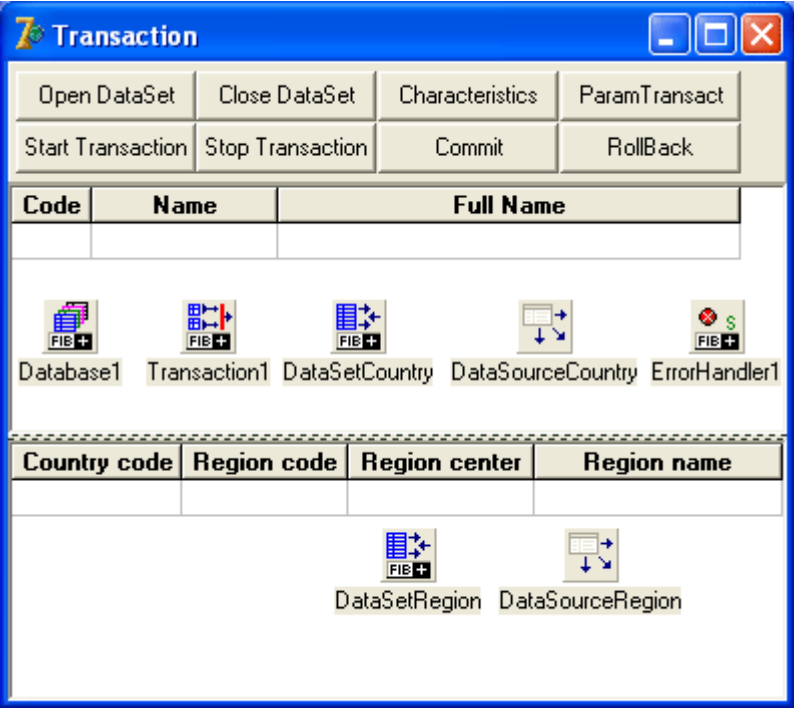
```
ON UPDATE CASCADE
);
COMMIT;
```

The tables have been filled with data about USA и ENGLAND countries and their regions.

Test application

Create a new application in Delphi or C++Builder IDE; create a toolbar with TButton; place two TDBDrid components TDBDrid: DBGridCountry and DBGridRegion; add two TDataSource components: DataSourceCountry and DataSourceRegion.

Then get the following components from FIBPlus tab: Database1 (TpFIBDatabase type), Transaction1 (TpFIBTransaction type), two TpFIBDataSet components: DataSetCountry and DataSetRegion; and the ErrorHandler component:



Picture 1. The main form of the test application

How to set necessary values for the database object:

Database name (DBName property) — FIBTRANSACT.FDB (put the database to the project directory).

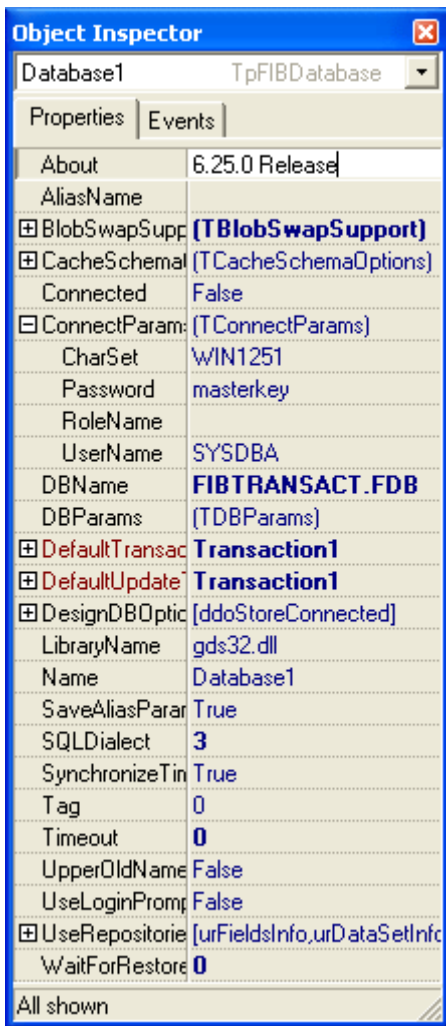
User name (UserName) SYSDBA,

Password (Password) masterkey,

CharSet (Charset) WIN1251,

Database SQLDialect (SQLDialect) equal to 3.

Select Transaction1 as DefaultTransaction and UpdateTransaction:

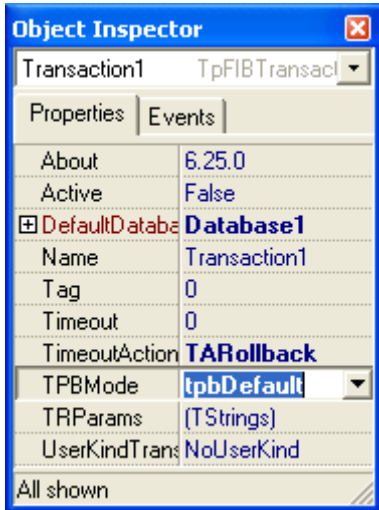


Picture 2. Database component properties

Set the following values for the Transaction1 component.

Select Database1 as a database name (DefaultDatabase) from the the popup menu.

Then select the tpbDefault value from the popup menu for the TPBMode transaction isolation level, as only this value will enable you to change the transaction parameter buffer (TPB). If you select tpbReadCommitted or tpbRepeatableRead values, corresponding constants will be placed to the parameter buffer on starting the transaction and you won't be able to change this.



Picture 3. Transaction component properties

For DataSetCountry dataset you can leave most properties as default. Set the following values:

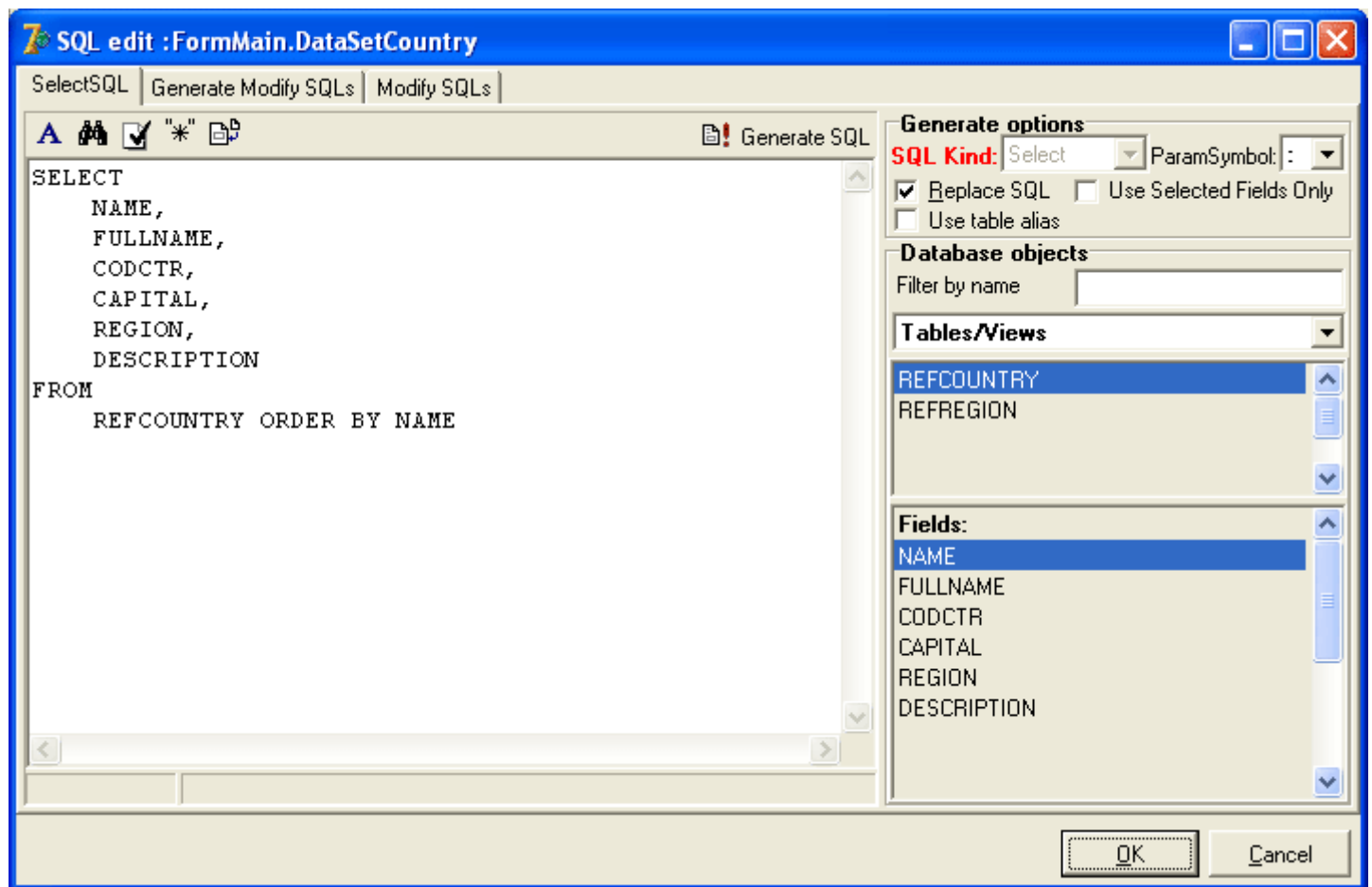
Database (Database) — Database1.

Transaction (Transaction) and UpdateTransaction (UpdateTransaction) — Transaction1.

Important: set poStartTransaction to False in the list of options (Option) in order to be able to manage transaction Start and Commit (or Rollback) explicitly. Set poKeepSorting to True. This setting can be useful for ordering datasets in case of data changes in ordered columns (ORDER BY clause).

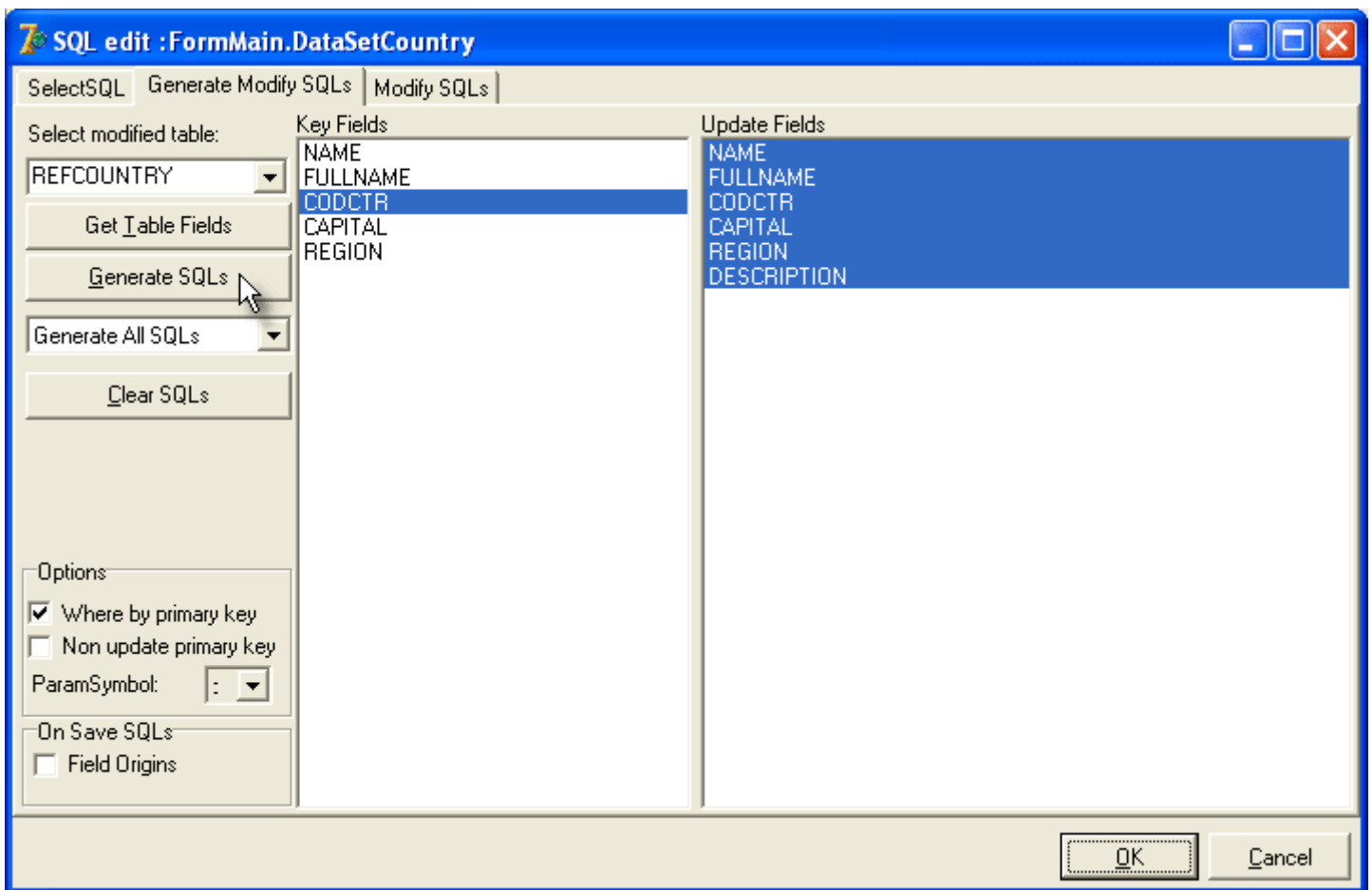
You can also set psAskRecordCount to True in PrepareOptions (not necessary in the demonstrated case). Use this option to show the number of retrieved records in the status bar after reopening the dataset.

Call the SQL generator (double click on the component and select SQL Generator string in the context menu). Select REFCOUNTRY from the list of tables and double click it. You will see the SELECT operator generated. Add ORDER BY NAME clause to the operator end to order the dataset by country names.



Picture 4. SELECT generation

Now click the Generate SQLs button on the Generate Modify SQLs tab. You will get Insert, Update, Delete and Refresh operators.



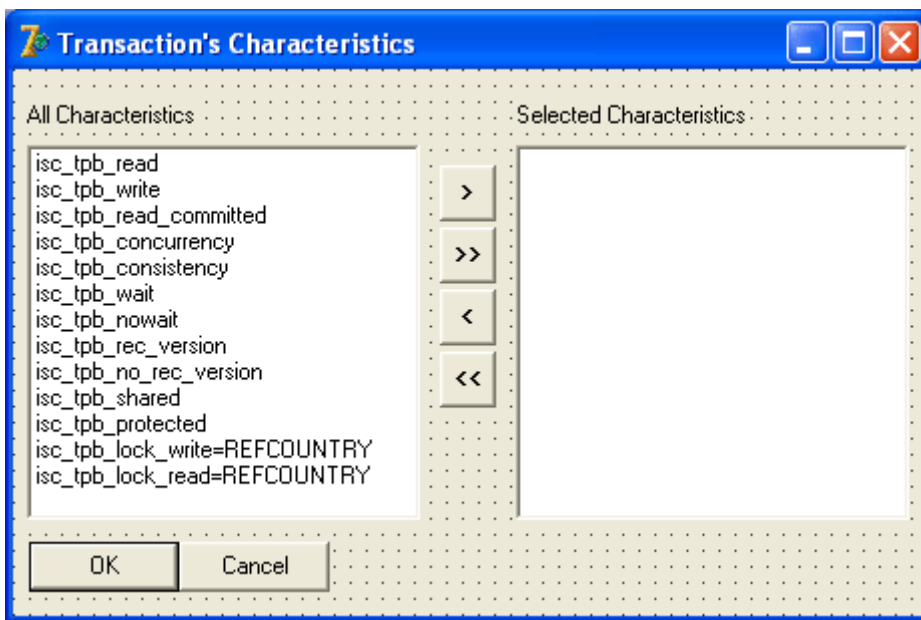
Picture 5. Generation of modifying operators

Link the DataSourceCountry component to the DataSetCountry dataset (the DataSet property), then do the same actions with the DataSetRegion component. Besides as this dataset is detail in the master-detail relationship you need to set DataSource to DataSourceCountry and also dcForceOpen and dcWaitEndMasterScroll to True in DetailConditions. On generating the SELECT operator in the SQL generator you should modify it:

```
SELECT
CODCTR,
CODREG,
CENTER,
REGNAME,
DESCRIPTION
FROM
REFREGION WHERE CODCTR = ?CODCTR
ORDER BY CENTER
```

From the list of regions the clause WHERE CODCTR = ?CODCTR selects only records which refer to the current country. More details about master-detail relationship in FIBPlus read at www.devrace.com.

Now create two forms: Transaction's Characteristics will form a list of transaction parameters - TRParams, whereas Transaction's Parameters will show the transaction parameter buffer (TPB).



Picture 6. Transaction's Characteristics Form

You will need the Transaction's Characteristics form to create a list of transaction characteristics. Set the Name property to AllParameters for the left ListBox component, and to SelectedParameters for the right ListBox. Now write the following TButton.OnClick event handlers. On clicking «>» selected strings are moved from the left ListBox to the right component.

```
procedure TFormTrans.Button1Click(Sender: TObject);
var I: Integer;
begin
  I := 0;
  while (I <= AllParameters.Items.Count - 1) do
  begin
    if AllParameters.Selected[I] then
    begin
      SelectedParameters.Items.Add(AllParameters.Items[I]);
      AllParameters.Items.Delete(I);
      I := I - 1;
    end;
    I := I + 1;
  end;
end;
```

Note. I do not write C++Builder code, as it's rather easy to translate from Delphi to C++.

You can select several strings in any ListBox component by pressing Ctrl and clicking the strings.

Click on the «>>» button to move all the strings from the left ListBox to the right. I have described this function just in case, as you don't need it for the present task.

```
procedure TFormTrans.Button2Click(Sender: TObject);
var I: Integer;
begin
  for I := 0 to AllParameters.Items.Count - 1 do
    SelectedParameters.Items.Add(AllParameters.Items[I]);
  AllParameters.Items.Clear;
end;
```

Click «<<» to move all selected strings from the right ListBox component to the left:

```

procedure TFormTrans.Button4Click(Sender: TObject);
var I: Integer;
begin
  I := 0;
  while (I <= SelectedParameters.Items.Count - 1) do
  begin
    if SelectedParameters.Selected[I] then
    begin
      AllParameters.Items.Add(SelectedParameters.Items[I]);
      SelectedParameters.Items.Delete(I);
      I := I - 1;
    end;
    I := I + 1;
  end;
end;

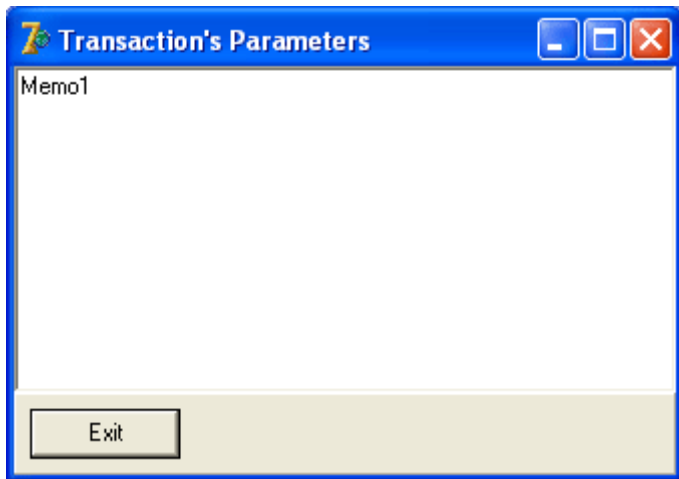
```

Click «<>» to move all the strings from the right ListBox component to the left:

```

procedure TFormTrans.Button3Click(Sender: TObject);
var I: Integer;
begin
  for I := 0 to SelectedParameters.Items.Count - 1 do
    AllParameters.Items.Add(SelectedParameters.Items[I]);
  SelectedParameters.Items.Clear;
end;

```



Picture 7. Transaction's Parameters Form

Transaction's Parameters Form shows the list of transaction parameters (TRParams) and transaction parameter buffer vector (TPB). On calling the form, Memo1 shows the parameter list and numeric values from TPB.

```

procedure TFormTransactionParam.FormShow(Sender: TObject);
var I: Integer;
begin
  Memo1.Clear;
  for I := 0 to FormMain.Transaction1.TRParams.Count - 1 do
    Memo1.Lines.Add(FormMain.Transaction1.TRParams.Strings[I]);
    Memo1.Lines.Add('=====');
  for I := 0 to FormMain.Transaction1.TPBLength - 1 do
    Memo1.Lines.Add(IntToStr(Integer(FormMain.Transaction1.TPB[I])));
  end;

```

Now return to the main module and write the form event handlers. Connect to the database in the OnShow form event and disconnect from the database in OnClose correspondingly:

Database1.Open; and

Database1.Close;

Click Open DataSet to open the DataSetCountry dataset and the DataSetRegion dataset will open automatically.

```
procedure TFormMain.BOpenDataSetClick(Sender: TObject);  
begin  
    DataSetCountry.Open;  
end;
```

Click Close DataSet to close both datasets:

```
DataSetCountry.Close;
```

Click BStartTransaction to start the transaction and BStopTransaction to stop it:

```
Transaction1.StartTransaction;  
Transaction1.Active := False;
```

You can commit the transaction by clicking BCommit. After Commit the transaction will be started again, then the dataset opens (it will be automatically closed after the transaction end by COMMIT or ROLLBACK):

```
procedure TFormMain.BCommitClick(Sender: TObject);  
begin  
    Transaction1.Commit;  
    Transaction1.StartTransaction;  
    DataSetCountry.Open;  
end;
```

The same goes for TButton.OnClick event by BRollBack.

```
procedure TFormMain.BRollBackClick(Sender: TObject);  
begin  
    Transaction1.Rollback;  
    Transaction1.StartTransaction;  
    DataSetCountry.Open;  
end;
```

Click Characteristics button to call Transaction's Characteristics Form and to form a user list of transaction characteristics.

```
procedure TFormMain.BCharactTransactClick(Sender: TObject);  
var I: Integer;  
begin  
    if FormTrans.ShowModal <> IDOK then exit;  
    if Transaction1.Active then  
        Transaction1.Active := False;  
        Transaction1.TRParams.Clear;  
    for I := 0 to FormTrans.SelectedParameters.Items.Count - 1 do  
        Transaction1.TRParams.Add(  
            FormTrans.SelectedParameters.Items[I]);  
end;
```

Click the ParamTransact button to call Transaction's Parameters Form and show current transaction characteristics.

Finally write the database exception handler. Then in case of exceptions you will be able to see SQLCODE and GDSCODE code values as well as database server message. For these purposes use the OnFIBErrorEvent event of the ErrorHandler component:

```
procedure TFormMain.ErrorHandler1FIBErrorEvent(Sender: TObject;  
ErrorValue: EFIBError; KindIBError: TKindIBError;  
var DoRaise: Boolean);  
var S: String;  
begin  
S := S + 'SQLCode = ' + IntToStr(ErrorValue.SQLCode) + #10#13;  
S := S + 'IBErrorCode = ' + IntToStr(ErrorValue.IBErrorCode) + #10#13;  
S := S + 'IBMessage = ' + ErrorValue.IBMessage + #10#13;  
Application.MessageBox(PAnsiChar(S), 'Database Error',  
MB_OK + MB_ICONSTOP);  
DoRaise := False;  
end;
```

Add fib to the list of program modules (uses clause).

Note. This exception handler shows only an exception message and does nothing else. You need to close the transaction manually, start it anew and open the dataset to continue working. Of course in real situations you will analyze the error and do everything to avoid it. In this case you will have a working application.

Part 2:

Transaction characteristics

All operations with a database (any changes of data, metadata, data selection, etc) are executed in the context of a transaction. All changes done in the context the transaction can be either committed (if there are no exceptions) or rolled back. You cannot commit the transaction if an exception error appears in any transaction operation, you can only roll back all the operations.

To start a transaction and set its characteristics the SET TRANSACTION operator is used in the SQL language. InterBase/Firebird API uses the function `isc_start_transaction()` and transaction parameter buffer TPB.

SQL COMMIT operator or its API equivalent `isc_commit_transaction()` are used for transaction committing, ROLLBACK or API `isc_rollback_transaction()` are used for rolling the transaction back.

FIBPlus uses corresponding API functions to work with a database server. To start a transaction with certain characteristics a developer should set the characteristics by any possible way and call the StartTransaction component or set the Active property to True.

Call the Commit method to commit the transaction and the Rollback method to roll it back. If you Commit/Rollback the transaction, all its datasets will be closed.

There are two more API functions and two FIBPlus methods of the transaction components: for committing the transaction with saving the context (the function `isc_commit_retaining()`, the CommitRetaining method) and for rolling the transaction back with saving the context (the function `isc_rollback_retaining()`, the RollbackRetaining method). These methods help developers to execute "soft" Commit/Rollback and make his life easier. Without them the programmer has to save the position of the current record (usually the primary key value is saved), restart the transaction, open a dataset and go to a necessary record. Using CommitRetaining and RollbackRetaining developers avoid doing these operations, but more server resources are used (it's a drawback). Besides in SNAPSHOT isolation level after "soft" Commit/Rollback the transaction won't see any changes executed by other processes.

InterBase/Firebird and FIBPlus also enable to create transaction save points and rollback to save points. I will consider these powerful features later.

To set characteristics from the transaction parameter buffer (TPB) mnemonic constants are used. Unfortunately constant names do not often correspond to SET TRANSACTION syntax elements.

Here is a simplified SET TRANSACTION syntax:

```
SET TRANSACTION
[READ WRITE | READ ONLY] /* access mode */
[WAIT | NO WAIT] /* permission locking mode */
[[ISOLATION LEVEL] /* isolation level */
{SNAPSHOT |
SNAPSHOT TABLE STABILITY |
READ COMMITTED [{RECORD_VERSION |
NO RECORD_VERSION}]]
[RESERVING <reserving clause>]
```

RESERVING clause sets the table reservation. The RESERVING clause syntax is:


```
<table> [, <table> ...]  
[FOR [SHARED | PROTECTED] {READ | WRITE}]  
[, <reserving clause> ...]
```

Default transaction value is (if there are no defined characteristics SET TRANSACTION or the transaction parameter buffer is empty):

```
SET TRANSACTION READ WRITE WAIT SNAPSHOT;
```

These default characteristics correspond to these TPB parameters:

```
isc_tpb_concurrency  
isc_tpb_write  
isc_tpb_wait
```

Using FIBPlus components you can form the transaction parameter buffer by placing a list of mnemonic constants defined in the module ibase.pas to the TRParams property of TpFIBTransaction.

The simplest characteristic is **access mode**. READ WRITE (isc_tpb_write в TPB) enables developers to read and modify database data in the context of this transaction. READ ONLY (isc_tpb_read в TPB) provides them with read only data operations.

The most important transaction characteristic is **isolation level**. The table below demonstrates three possible isolation levels.

Transaction isolation levels in InterBase/Firebird.

SQL	TPB Constant	Value
READ COMMITTED	isc_tpb_read_committed	<p>Reading of committed changes. The transaction can see the latest committed database changes done by other transactions.</p> <p>This isolation level has two opposite parameters:</p> <p>Default NO RECORD_VERSION (isc_tpb_no_rec_version in TPB) requires Commit of all data changed by other transactions.</p> <p>RECORD_VERSION (isc_tpb_rec_version in TPB) enables you to read the latest version of committed changes, even if there are other uncommitted changes.</p>
SNAPSHOT	isc_tpb_concurrency	<p>Snapshot is a default setting; its other name is Repeatable Read. It shows the database snapshot before the transaction start. This transaction sees no changes executed by other transactions, only its own changes.</p>
SNAPSHOT TABLE STABILITY	isc_tpb_consistency	<p>It is an isolated snapshot, Serializable. It is similar to SNAPSHOT, the only difference is that other transactions can read data from tables of this transaction, but they cannot make any changes.</p>

Besides the isolation levels listed in the tables there is another level Dirty Read or Read Uncommitted, described in books on databases. This isolation level helps to read uncommitted changes done by other transactions. InterBase and Firebird do not support Dirty Read.

One more important transaction characteristic is **the locking permission mode**. If WAIT (isc_tpb_wait) is set to True and a conflict arises, the transaction will wait for locking permission from other transactions by giving them Commit or Rollback operators. If NO WAIT (isc_tpb_nowait) is set to True, in case of locking this transaction shows an exception and forms code values. WAIT is set by default.

I will provide you with more details about **reserving** in the corresponding part of the article.

If you set the TPBMode transaction property to tpbReadCommitted (READ COMMITTED isolation level; this value is set for the transaction when you put a component onto the form), after the transaction start this property will have the following constants (regardless the values you define in TRParams):

```
isc_tpb_write  
isc_tpb_nowait  
isc_tpb_rec_version  
isc_tpb_read_committed
```

Similar, if you set TPBMode to tpbRepeatableRead (SNAPSHOT isolation level), TRParams will have:

```
isc_tpb_write  
isc_tpb_nowait  
isc_tpb_rec_version
```

Besides this will also be a default setting:

```
isc_tpb_concurrency
```

So if you want to manage transaction characteristics directly from the application, set tpbDefault to True in TPBMode.

Note: the first parameter sent to TPB is always isc_tpb_version3, which sets the transaction version. You do not need to send this parameter when using FIBPlus, because it is formed and sent to API function isc_start_transaction() automatically. Working with the application click ParamTransact and you will see the form with both mnemonic names of transaction parameters (which you have set in TRParams) and real numbers in TPB. Always the first number will be 3 - it is the value of isc_tpb_version3 parameter.

Here are some numeric parameter values used to form TPB. I will not consider several of them in this article. The parameters are defined in ibase.pas.

```
isc_tpb_version1 = 1;  
isc_tpb_version3 = 3;  
isc_tpb_consistency = 1;  
isc_tpb_concurrency = 2;  
isc_tpb_shared = 3;  
isc_tpb_protected = 4;  
isc_tpb_exclusive = 5;  
isc_tpb_wait = 6;  
isc_tpb_nowait = 7;  
isc_tpb_read = 8;  
isc_tpb_write = 9;  
isc_tpb_lock_read = 10;  
isc_tpb_lock_write = 11;  
isc_tpb_verb_time = 12;  
isc_tpb_commit_time = 13;  
isc_tpb_ignore_limbo = 14;  
isc_tpb_read_committed = 15;  
isc_tpb_autocommit = 16;  
isc_tpb_rec_version = 17;  
isc_tpb_no_rec_version = 18;  
isc_tpb_restart_requests = 19;
```

```
isc_tpb_no_auto_undo = 20;  
isc_tpb_last_tpb_constant = isc_tpb_no_auto_undo;
```

Experimenting with transaction characteristics

Let's start experiments with transaction characteristics. Notice that the program has so called "long" transaction: a user manually starts the transaction, does different data changes and commits/rolls the transaction back whenever necessary. This may cause locking. An example of a "short" transaction is: the user presses OK on the data adding/editing form; the application calls Insert or Edit for the corresponding dataset; sets column values and calls the Post method, which sends changes to the database; Update transaction is started on calling Post; after Post the transaction is committed. As a rule such a short transaction lives less than a millisecond, thus less locking possibility occurs in multi-user database environment. I would like to investigate on which conditions a locking occurs, so I will use long transactions.

READ COMMITTED isolation level

READ COMMITTED isolation level is used most often. It enables transactions to read committed changes executed by other transactions. I will demonstrate you how to use this level. Run two copies of the test application. Set the following transaction characteristics clicking the Characteristics button in the first application:

```
isc_tpb_write  
isc_tpb_read_committed  
isc_tpb_rec_version  
isc_tpb_nowait
```

This means that the transaction can read and write data (isc_tpb_write) and has the isolation level READ COMMITTED (isc_tpb_read_committed), i.e. the transaction reads all committed changes done in other transactions. The transaction will be "happy" with any latest committed version of the record, even if there other uncommitted version of the same record (isc_tpb_rec_version). If a locking conflict occurs the transaction won't wait for the conflict solution from another transaction but shows an exception (isc_tpb_nowait).

Set the following characteristics for the second transaction:

```
isc_tpb_write  
isc_tpb_read_committed  
isc_tpb_rec_version  
isc_tpb_wait
```

The only difference from the first transaction is the last string. In case of the locking conflict the second transaction will wait for the solution from another transaction.

In any isolation levels the locking conflict always occurs when two transactions try to edit the same able record.

Ready? Now I will create conflict situations and solve them.

Start a transaction in the first application, open a dataset and change some column value in DBGridCountry in the region reference. Then after making the changes click/navigate to any other table record and make it current. The Post operation will send the changes to the database.

Do the same sequence of actions in the second application: start the transaction, open the dataset, change the same record (as in the first application), you can change the value of any record column. Make another record current to send the changes to the database.

What has happened? The second application is out of order: it's good if you can see the mouse cursor, in some cases it's not seen at all. The point is that you have set (isc_tpb_wait) mode for the second application (it is waiting for the locking conflict solution). Click to commit the transaction in the first table, the second application will suppress a corresponding error message (your event handler has done this).

Repeat the same actions: change a record in the first program, change the same record in the second application. The second program will start waiting. Roll back the transaction in the first program, and the changes from the second program will be sent to the database without errors.

Conclusion 1. Do not use isc_tpb_wait mode when using long transactions.

Change the waiting mode in the second application: remove the isc_tpb_wait parameter from the transaction characteristics and add isc_tpb_nowait.

Note! isc_tpb_wait is used by default. You should always set isc_tpb_nowait explicitly.

A small example from my practice: when I was preparing a demo example to show how **short** transactions behave well in multi-user environment (see "Advantages of using FIBPlus components"), I made a simple mistake and created a **long** transaction instead (I had not set values of two properties). 11 students of mine changed the same record on client machines and went to another record to send the changes to the server. If I had had a short transaction, the locking could have occurred only on a couple of machines. In my case the long transaction caused an error exception at ten computers and only the 11th could update the changes well.

Now I will show another conflict. Delete USA record from the first application by pressing Ctrl+Del; confirm the delete operation in the standard dialog form. Do not commit the transaction. Delete the same USA record from the second program. You will get the conflict. Roll back the transaction in both cases. The deleted record will appear in the list again. Again delete it from application 1. Try to change or delete any record in region tables (DBGridRegion) of the same country in the second program. Another conflict occurs. It will also occur when you try to change the key column (country code) in the first program and then to change/delete the detail region record in the second. In the region table the CodCtr column is a foreign key, which refers to the country code in the country table. The foreign key description is:

```
CONSTRAINT "Region_FOREIGN_KEY"  
FOREIGN KEY (CodCtr) REFERENCES REFCOUNTRY (CodCtr)  
ON DELETE CASCADE  
ON UPDATE CASCADE
```

This means that on changing the primary key value of the master table the detail foreign key value will be also changed.

In other words conflicts will always occur when you try to change (edit or delete) any record of any table (master or detail) involved into the changing process in another transaction.

Note. Notice that the restriction name is set explicitly in the foreign key description. Actually it is always good to name all column and table restrictions, as you will get a more clear error message from the database server in case of the restriction violation.

Now let's do another experiment. Close the dataset in the second program. Change a record without committing the transaction in the first program. Open the dataset in program 1. As you see everything is ok, the second transaction sees the old unchanged record version. Now change characteristics of the second transactions. Remove its isc_tpb_rec_version parameter and set it to isc_tpb_no_rec_version. Start the transaction in the second program and try to open the dataset. You will get an exception.

It happens because of the `isc_tpb_no_rec_version` parameter. It requires committing of all changes of data used in tables of your transaction and modified by other transactions. The case is even worse: if you change a record in the first program without committing the transaction, roll back the transaction in the second application and then try to reopen the dataset, you will also get an exception.

Cconclusion 2. Do not use `isc_tpb_no_rec_version` together with `c isc_tpb_nowait` if you have long transactions or many clients who work with you database.

Note! The `isc_tpb_no_rec_version` parameter is used by default. You should set `isc_tpb_rec_version` explicitly.

Now try to set `isc_tpb_no_rec_version` together with `c isc_tpb_wait`, and you will get very interesting results. Of course you remember that if you set `isc_tpb_rec_version` with `isc_tpb_wait`, in case of conflict the second program will be in the waiting mode; if the transaction is committed in the first program, the second will show an exception.

If you set `isc_tpb_no_rec_version` and `isc_tpb_wait`, the second program shows an exception neither after rolling back the transaction nor after committing it in the first program. This feature enables to considerably decrease or even avoid locking conflicts in a multi-user environment. I will demonstrate you in more detail how to use such transaction characteristics when describing separate transactions.

Let's have a look at how to set the `isc_tpb_read` parameter. Set it for a transaction after removing `isc_tpb_write` and try to change the data. You will get an exception. The transaction with `isc_tpb_read` really does not allow you to do data changes.

It's time to show dead lock (mutual lock) situation: both programs will be waiting for Commit/Rollback from each other.

Set the following transaction characteristics in both applications:

```
isc_tpb_write  
isc_tpb_read_committed  
isc_tpb_wait  
isc_tpb_rec_version
```

Star the transactions and open datasets. Change a record in application 1 and another (different) record in application 2. Then in application 2 try to change the same record, which was modified in the first program. At this moment application 2 will be in the waiting mode as it has `isc_tpb_wait` transaction parameter (which waits for locking conflict solution from another transaction). Now in application 1 change the record, which was modified by the second application. The first application also starts waiting and deadlock occurs, both applications are waiting for each other.

Actually nothing serious has happened. In a few seconds the first application will show an exception message: "deadlock. update conflicts with concurrent update". The database server has features responsible for mutual locks. Lock Manager analyses locks not permanently but from time to time (in a certain period of time) to increase application performance. The interval for Lock Manager is set by the `DeadlockTimeout` parameter in `firebird.conf` for Firebird 1.5 or by `DEADLOCK_TIMEOUT` in `ibconfig` for InterBase. The default interval value is 10 seconds.

Now do the last experiment. Roll back both transactions and reopen the datasets in both programs. Change a record in the first program and commit the transaction. Change **the same** record in the second application and also do Commit. This operation is a success. Now reopen the dataset in both applications. The database keeps changes **only of the last operation**.

Reminding. To make the transaction with READ COMMITTED see committed changes of other transactions you only need to reopen the dataset (FIBPlus usually uses the FullRefresh method). For this you don't need to stop and restart the transaction.

SNAPSHOT isolation level

SNAPSHOT isolation level is set by default. It helps to read unchanged database state on transaction starting. In this transaction you cannot see changes done by other transactions; this transaction sees only its own changes.

It the first application set the following parameters for READ COMMITTED isolation level:

```
isc_tpb_write  
isc_tpb_read_committed  
isc_tpb_rec_version  
isc_tpb_nowait
```

For the second application set:

```
isc_tpb_write  
isc_tpb_concurrency  
isc_tpb_nowait
```

Change a record in the first program without committing the transaction. Try to modify the same record in application 2 and you will certainly get the lock exception. Now change any record in application 1 and commit the transaction. Close the dataset in application 2 and reopen it. The second application with SNAPSHOT does not see modifications committed in the other transaction. That's correct, because SNAPSHOT isolation level shows the database snapshot made at the transaction start, and this snapshot cannot be changed by any concurrent transactions. Changes made by other transactions can be seen in application 2 after you stop the transaction, restart it and open the dataset.

One more interesting test: modify any record and commit the transaction in the first program. Try to change the same (committed!) record in the second application (it does not see changes of the first program). You will also get an exception. I suspect many users will be surprised by such application behaviour.

This is almost everything I wanted to tell about SNAPSHOT isolation level. Use it when you either want to know nothing about the external situation (which changes were made after your transaction started) or want to get uncontrolled exceptions.

SNAPSHOT TABLE STABILITY isolation level

It is an isolated, serializable snapshot. It is almost similar to SNAPSHOT, the only difference is that other transactions regardless their isolation levels can only read data from tables of this transaction without modifying them.

Modify transaction characteristics in the second application; make the following list of parameters:

```
isc_tpb_write  
isc_tpb_nowait  
isc_tpb_consistency
```

Start the transaction and open the dataset.

In the first application try to change or delete data in master or detail table and you will get an exception. SNAPSHOT TABLE STABILITY enables you to work with the table on your own; nobody else can do the same simultaneously with you. There is one not obvious problem: if the transaction with SNAPSHOT TABLE

STABILITY is started and another transaction has already changed the data without committing them, dataset opening will cause an exception.

Let's check this in practice. Stop the transaction in application 2. Change a record in the country table in the first application. Now you can do this because the second transaction is not active. Then start the transaction in application 2 and try to open the dataset. You will see an exception message "lock conflict on no wait transaction". Stop the transaction in the second application, commit/rollback the changes in the first program and open the dataset in the second program.

See what will happen when after seeing this exception at first you commit changes in the first program and then click OK in the exception message of the second application. You will see the dataset **without** changes, executed and not committed by the first application at the moment of the second application start! As you understand, you should be very cautious when writing event handlers.

Part 3:

How to reserve tables

SNAPSHOT TABLE STABILITY isolation level reserves tables used in such transaction. There additional features enabling you to reserve tables in the transaction or vice versa to allow other transactions modify tables in SNAPSHOT TABLE STABILITY isolation level. In the SQL language it is a reserving operator, in TPB these are constants `isc_tpb_lock_read`, `isc_tpb_lock_write`, `isc_tpb_exclusive`, `isc_tpb_shared`, `isc_tpb_protected`. Here are examples on how to use them in practice.

Editing tables in READ COMMITTED isolation level

Run two copies of the application. In the first application set the following transaction parameters:

```
isc_tpb_write
isc_tpb_read_committed
isc_tpb_nowait
isc_tpb_rec_version
isc_tpb_lock_write=REFCOUNTRY
isc_tpb_protected
```

In SQL it will be:

```
SET TRANSACTION
READ COMMITTED READ WRITE
NO WAIT
RECORD_VERSION
RESERVING REFCOUNTRY FOR PROTECTED WRITE;
```

Note! The `isc_tpb_protected` parameter must be just after the name of the reserved table.

For the second application set a standard READ COMMITTED transaction:

```
isc_tpb_write
isc_tpb_read_committed
isc_tpb_nowait
isc_tpb_rec_version
```

Start the transactions and open datasets in both applications. Then try to change any record in the second application. You will get a lock exception because REFCOUNTRY is reserved.

Now stop the transaction in the applications and close the datasets. Start the transaction in application 1 without opening the dataset. Start the transaction in application 2, open the dataset and try to modify it. Again an exception appears. You can see that the tables are being reserved on the transaction start, not on the dataset opening (like in SNAPSHOT TABLE STABILITY).

In the first transaction replace `isc_tpb_protected` by `isc_tpb_exclusive`. The result will be the same. Change `isc_tpb_lock_write=REFCOUNTRY` to `isc_tpb_lock_read=REFCOUNTRY`, the result is the same again. If you set the `isc_tpb_shared` parameter, no reserving will be in READ COMMITTED isolation level.

The same results will be obtained if you remove SNAPSHOT and SNAPSHOT TABLE STABILITY isolation levels in the second application. In the last case you also won't be able to modify the data in application 1.

So reserving for READ COMMITTED transaction helps to prevent the reserved table from changes by another transaction with any isolation level.

Reserving tables in SNAPSHOT isolation level

Set these transaction characteristics for the first application:

```
isc_tpb_write  
isc_tpb_concurrency  
isc_tpb_nowait  
isc_tpb_lock_read=REFCOUNTRY  
isc_tpb_exclusive
```

Do the same tests as with READ COMMITTED. You will get absolutely similar results.

Reserving tables in SNAPSHOT TABLE STABILITY isolation level

SNAPSHOT TABLE STABILITY isolation level does not allow other transactions to modify data in used tables. But reserving means can allow changes for certain tables. Set the transaction characteristics in the first program:

```
isc_tpb_write  
isc_tpb_nowait  
isc_tpb_lock_write=REFCOUNTRY  
isc_tpb_shared  
isc_tpb_consistency
```

If the second transaction is READ COMMITTED or SNAPSHOT, it can change data in the defined table. But of course it's still not possible to change data in the same record simultaneously.

Start the transactions and open the datasets. You can modify the table with (isc_tpb_shared) in the second application.

Reminding. Parameters isc_tpb_shared, isc_tpb_exclusive and isc_tpb_protected must be placed just after isc_tpb_lock_write (isc_tpb_lock_read). Any table name must be mentioned in reserving operators not more than once.

Using separate transactions

FIBPlus components enable developers to use two transactions for work with datasets: one for reading and the other for updating/writing. I will demonstrate you which transaction characteristics are possible.

Create a new project Transaction2 based on the current project. Make some changes. Add another transaction component and name it Transaction2. It will be an updating/writing transaction. Set Transaction2 as updating in the database component and in both datasets (the DefaultUpdateTransaction property in Database; UpdateTransaction in datasets). Set AutoCommit and poStartTransaction (in Options) to True in the datasets. It is a short updating transaction. It will be started every time after Post (when the application sends changes to the database) and automatically committed in case of no exceptions.

Set TPBMode to tpbDefault in the second transaction and the following values in TRParams window:

```
isc_tpb_write  
isc_tpb_read_committed  
isc_tpb_nowait  
isc_tpb_rec_version
```

Note. This slightly modified test application is shown in the project Transaction2.

Run the application and set characteristics for the first transaction:

```
isc_tpb_read  
isc_tpb_concurrency  
isc_tpb_nowait
```

These characteristics correspond to the SNAPSHOT isolation level. Start the transaction, open the dataset and change any record. Click any other record to do Post and Commit for the modified record. The record will still have the old unchanged column value! That's correct; the reading transaction Transaction1 sees no changes executed by the updating transaction Transaction2 in the same application. Even if you close the dataset and reopen it, nothing will change. You need to stop the transaction, restart it and open the dataset, and only then you will see a new changed column value. The reading transaction sees no changes done by the other transaction even if both of them are in the same application.

I have demonstrated you a bad variant, but it can be even worse. Create the following characteristics for the reading transaction:

```
isc_tpb_read  
isc_tpb_nowait  
isc_tpb_consistency
```

The reading transaction will have SNAPSHOT TABLE STABILITY isolation level, so other transaction cannot change its tables. Start the transaction, reopen the dataset and try to change a record. You will see lock conflict.

Of course these are normal parameters for the reading transaction:

```
isc_tpb_read  
isc_tpb_nowait  
isc_tpb_read_committed  
isc_tpb_rec_version
```

Besides the reading transaction must do no table reserving. As for the updating/writing transaction, you can choose any necessary isolation level and additional characteristics.

If you have a separate long reading transaction (with READ ONLY or isc_tpb_read mode), you will considerably save database server resources.

Advantages of using FIBPlus components

You have spent much time and many efforts to get locks in applications which work with the same database. Now I will consider all FIBPlus advantages to help you to create "correct" programs and decrease lock possibility.

Using separate transactions correctly

Run the project Transaction2. Set these parameters for the reading transaction (Transaction1):

```
isc_tpb_read  
isc_tpb_read_committed  
isc_tpb_nowait  
isc_tpb_rec_version
```

This is a long read-only transaction with READ COMMITTED. One of its advantages is minimal use of server resources.

The updating transaction (Transaction2) is short. It is started on calling the Post method, which posts changes to the server, and it is committed automatically after Post (if no Update exceptions occur). You can hook the transaction start and commit by writing event handlers for Transaction2 transaction component: AfterStart or BeforeStart to hook the start and AfterEnd (BeforeEnd) to close the transaction. The test application will show corresponding messages for these events (you can hide comments in real applications).

Run the second application and set the same parameters for the reading transaction. Now try to get a lock by changing the same record in both applications. Most likely you won't be able to get a lock on one local machine due to the short update transaction. Many times 11 students and I practiced a lot, trying to simultaneously change the same record in the server database in the local network and to move to a new record together to commit the transaction. In all the cases we got from 0 to 3 lock conflicts.

You can absolutely avoid update conflicts when using separate transactions. Set these parameters for the updating transaction:

```
isc_tpb_write  
isc_tpb_read_committed  
isc_tpb_wait  
isc_tpb_no_rec_version
```

In case of lock conflict with the concurrent process the transaction will be in the waiting mode. It won't last for a long time as the concurrent updating transaction is short. After the concurrent transaction is rolled back/committed, our process will be executed. The database will keep only the last data modification. We checked this situation on the 11 computers in the network and got no conflict.

It's up to you to decide whether it is good or bad, it depends on a certain task you need to do.

Note. Of course if the conflict has occurred when one process deleted a record and the other tried to change this deleted record, the exception message will be shown.

So if you know how to use separate transactions correctly, you will have far fewer lock exceptions.

Using poProtectedEdit mode

Now return to the first project. Set poProtectedEdit to True in the Option property in the DataSetCountry dataset. This will help you to lock a record which will be modified.

Run the two programs. Set READ COMMITTED isolation level for both transactions. Try to change a record in the first application without committing the transaction. Then modify the same record in the second application.

You will certainly get a lock conflict, but the exception message will appear at once, when you try to commit at least a record symbol, not after the second application tries to commit the transaction (as you've seen in our first experiments). Due to this feature you can avoid the following annoying situation: a user is modifying a database record with many columns for a long time, tries to send the changes to the database and suddenly gets to know that the record could not be modified as another user has been modifying the same record.

Due to the mechanism of dummy updates FIBPlus helps you to avoid this situation: to reserve only one record and forbid its concurrent changes, FIBPlus shows the UPDATE operator. It changes nothing in the record (i.e. sets the same value to one column). The server creates a new record version which does not differ from the last committed version and locks this record. In this case concurrent users cannot change this record until the transaction is committed.

Note. In this way you can lock as many records as necessary. When you start changing a new record in the context of one transaction, this record becomes protected/locked. There are no other ways to cancel the record locking but to commit/roll back the transaction.

Part 4:

Working with transactions in multiple databases

InterBase and Firebird servers enable developers to use a single transaction while working with multiple databases. Two-phase commit in limbo transactions and other similar features are described in different articles about transactions, so I will not consider this topic in detail.

FIBPlus components have full support of transactions in multiple databases. Besides, the library has special components, which enable robust work with several databases. In the example below I will use the UpdateObject project (with slight changes):

There are two databases: COUNTRY.GDB and PERSON.GDB. COUNTRY.GDB has a country reference REFCOUNTRY, PERSON.GDB has the PERSON table:

```
CREATE TABLE PERSON (  
    CODPERS INTEGER NOT NULL,  
    FIRST_NAME VARCHAR(20),  
    LAST_NAME VARCHAR(20),  
    COUNTRY CHAR(3),  
    PRIMARY KEY (CODPERS)  
);
```

I create the GEN_PERSON generator to get a value of the artificial primary key CODPERS. This is a trigger which fills in the primary key with a generator value::

```
CREATE TRIGGER CREATE_PERSON FOR PERSON  
ACTIVE BEFORE INSERT POSITION 0  
AS BEGIN  
    IF (NEW.CODPERS IS NULL) THEN  
        NEW.CODPERS = GEN_ID(GEN_PERSON, 1);  
    END
```

The table has 22 records from a demo database EMPLOYEE.GDB.

Note: the record has a column - the COUNTRY code. The column is a foreign key, which refers to a country code in REFCOUNTRY (column CODCTR) placed in another database COUNTRY.GDB. No links between tables in different databases are possible at the data structure level described by SQL means. So I cannot write FOREIGN KEY there and use database server means for data link integrity. I also cannot write a trigger or a stored procedure to do these actions as both of them (the trigger and SP) can work only with their "native" database. So I should find my own way out.

Luckily FIBPlus has necessary means, so I will consider them now.

At first I will create a new project MultiBase. Then I'll place two DBGrid components on the form: one for the country reference and the other for the list of employees; set the Align property and put Splitter. After this I'll drop two TpFIBDatabase components, two TpFIBTransaction components (one for reading, one for writing), two datasets and two DataSource components. Then I'll set the DefaultTransaction property to ReadTransaction in both TpFIBDatabase components (DefaultTransaction is a default transaction), and the DefaultUpdateTransaction to WriteTransaction (DefaultUpdateTransaction is a default Update transaction). As a result both databases will be in the internal database list of each transaction. The transactions will start in both databases at once. If I place 10 database components on the form and set default transactions, each of them will be started in the ten databases.

To check this fact I will add another menu item: Transact Databases. Database names for both transactions can be seen on calling this item:

```
procedure TFormMain.MTransactDatabasesClick(Sender: TObject);  
var i: integer;  
    S: String;  
begin  
    S := 'WriteTransaction' + #10#13;  
    S := S + 'DatabaseCount: ' +  
    IntToStr(WriteTransaction.DatabaseCount) + #10#13;  
    for i := 0 to WriteTransaction.DatabaseCount - 1 do  
    S := S + WriteTransaction.Databases[i].Name + #10#13;  
    S := S + #10#13 + 'ReadTransaction' + #10#13;  
    S := S + 'DatabaseCount: ' +  
    IntToStr(ReadTransaction.DatabaseCount) + #10#13;  
    for i := 0 to ReadTransaction.DatabaseCount - 1 do  
    S := S + ReadTransaction.Databases[i].Name + #10#13;  
    Application.MessageBox(PChar(S), 'ReadTransaction', MB_OK);  
end;
```

If you add new database components and set default transactions for them, you will see these databases included into the transaction list.

In the PersonData component I'll set automatic transaction start and automatic transaction commit (and I won't set these options in CountryData) writing the SQL operators. Then the components should be linked with each other by means of corresponding property values.

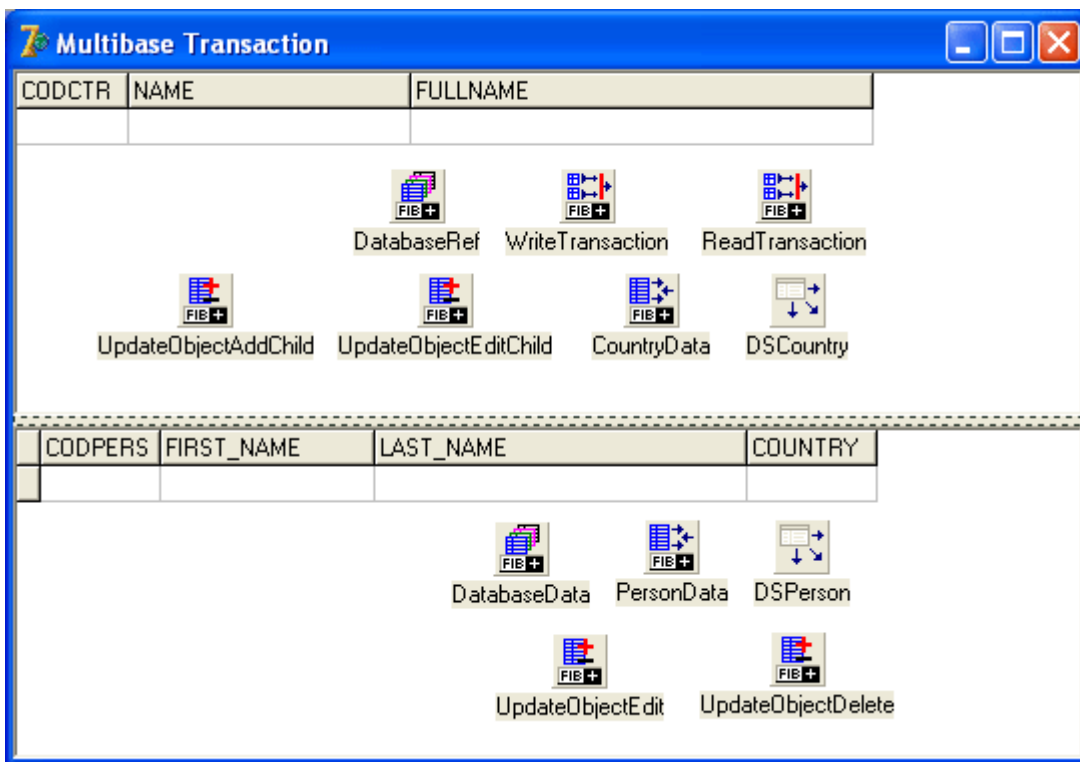
Let's place a menu onto the form and create two items: Commit and Rollback, which will commit the transaction and roll it back:

```
procedure TFormMain.MCommitClick(Sender: TObject);  
begin  
    WriteTransaction.CommitRetaining;  
    PersonData.FullRefresh;  
end;  
procedure TFormMain.MRollbackClick(Sender: TObject);  
begin  
    WriteTransaction.RollbackRetaining;  
    CountryData.FullRefresh;  
end;
```

Commit makes sense when I am editing the country reference (on editing employees Commit is automatic). To show changes of Employees records in DBGrid I call the FullRefresh dataset method.

Similarly when I am editing countries, new (changed) records are shown in the list. On rolling back the transaction I need to read the countries anew in order to return to the initial record variant.

In the OnShow event I will write commands to connect transactions, start transactions and open both datasets. On closing the form I close the transactions, commit/roll them back and disconnect.



Picture 8. Project MultiBase

Everything works well: both transactions are started for both databases. And I can view and edit both tables independently. But actually we need to create links between these tables placed in different databases.

I will use the TpFIBUpdateObject component in FIBPlus: place two TpFIBUpdateObject components (UpdateObjectEdit and UpdateObjectDelete) onto the form and use them to bring the PERSON table to conformity with REFCOUNTRY. They are responsible for foreign key functions: ON UPDATE CASCADE and ON DELETE CASCADE.

In UpdateObjectEdit I'll set the Database property to DatabaseData. This means that the SQL operator of this component works with the database defined in DatabaseData (the database PERSON.GDB) and its SQL operator will refer to the table in that database.

Then set the DataSet property to CountryData by selecting it in the pop-up list. So you have defined the dataset name, and the component will react at one of the dataset events. The dataset refers to a table placed in another database. I'll select some event from the pop-up list of the KindUpdate: ukModify property. This means that the SQL command of the component will be executed after the data in CountryData will be changed. I need to set the transaction for the WriteTransaction component. This is a transaction where the SQL command of the UpdateObjectEdit component will be executed.

Write the following statement in the SQL property:

```
UPDATE PERSON SET COUNTRY = :CODCTR
WHERE COUNTRY = :OLD_CODCTR
```

How does it work? If a user changes any string in REFCOUNTRY and commits the Update transaction (by clicking any other table record), the SQL command of the component starts. It replaces all code country values (COUNTRY) in PERSON by a new value taken from REFCOUNTRY (the parameter :CODCTR). Changes will be applied only to records which country code equals to the old country code value in REFCOUNTRY (the parameter :OLD_CODCTR).

So you have imitated the ON UPDATE CASCADE statement in the foreign key description.

In the same way set the UpdateObjectDelete properties. The only difference is that you should select the ukDelete value in the KindUpdate property and write the following code in the SQL property:

```
DELETE FROM PERSON
WHERE COUNTRY = :OLD_CODCTR
```

Select the same code as for UpdateObjectEdit in AfterExecute event handler.

Then run the application. Change any country code in the country reference and move to another string to send the changes to the server. As the transaction has not been committed, you cannot see any changes in the second table (the writing transaction sees only committed changes). Commit the transaction in the corresponding menu item. Now you will see that the country codes have changed in all necessary tables in PERSON. Delete some country (Ctrl+Del and OK in the dialog window). Commit the transaction and all PERSON records referring to this country will be deleted.

You can do as many changes and deletes as you need. The personnel list will be changed only after the transaction is committed.

So you have imitated the foreign key behaviour (one way).

Now you need to check how to add new records into PERSON correctly and change foreign key values in this table (the country code). You need to forbid changing or adding this column if a country table lacks for some record with the corresponding primary key. As the foreign key can have a NULL value, you shouldn't check Null values of the column.

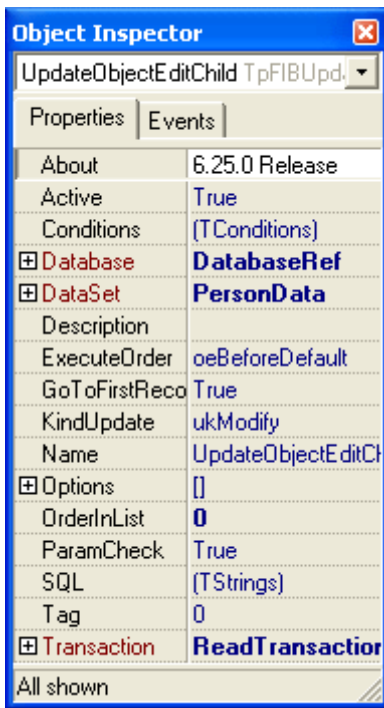
Create an exception and a stored procedure to check Null values in COUNTRY.GDB:

```
CREATE EXCEPTION NO_COUNTRY
  'There is no country with such a code in the country reference';
COMMIT;

SET TERM !! ;

CREATE PROCEDURE TEST_COUNTRY (CODCTR CHAR(3))
AS
DECLARE VARIABLE COUNTRY_NUM integer;
BEGIN
  if (:CODCTR != '') then
  begin
    select count(*) from REFCOUNTRY where CODCTR = :CODCTR
    INTO :COUNTRY_NUM;
    if (:COUNTRY_NUM = 0) then
      EXCEPTION NO_COUNTRY;
    end
  end
END !!
SET TERM ; !!
COMMIT;
```

Place two components: UpdateObject: UpdateObjectAddChild and UpdateObjectEditChild onto the form. The UpdateObjectEditChild properties are shown in the picture.



Picture 9. UpdateObjectEditChild component properties

The SQL property refers to the stored procedure:

```
EXECUTE PROCEDURE TEST_COUNTRY (:CODCTR)
```

The UpdateObjectAddChild component has the same properties except for KindUpdate (it has the ukInsert value).

Write the BeforePost: event handler for the PersonData dataset:

```
procedure TFormMain.PersonDataBeforePost(DataSet: TDataSet);
begin
  UpdateObjectAddChild.ParamByName('CODCTR').AsString :=
  PersonData.FieldByName('COUNTRY').AsString;
  UpdateObjectEditChild.ParamByName('CODCTR').AsString :=
  PersonData.FieldByName('COUNTRY').AsString;
end;
```

Here we form CODCTR parameter values of UpdateObject components.

If a new record has been added to PERSON or a value of the current country code value has not been found in the country reference, the exception "There is no country with such a code in the country reference" will arise.

Run the application and in PERSON change the country code to some non existing value. You will get this exception. If you delete the country code value (which will get a NULL value), nothing will happen. This is the behaviour we wanted to have.

Note. This was only a demonstration example showing how FIBPlus can work with multiple databases. In practice you need to optimize your code and thus set more options, in particular, use other component events, write your own exception handler, etc.

Nested transactions

InterBase and Firebird have nested transactions, which are also called user savepoints.

In case you have a long transaction savepoints enable you to create points which fix the current database state. The application names these savepoints. Looking at the savepoints you can know the database state at any moment (before the transaction commit or rollback). In this case previous savepoints remain whereas the current savepoint (which was rolled back) and all consequent savepoints are cancelled.

Actually everything is simple. To create a savepoint use the SQL SAVEPOINT statement:

```
SAVEPOINT <identifier>;
```

The identifier is any correct database object name shorter than 31 symbols. If there is already a savepoint with the existing name, it will be replaced by a new savepoint. That means that the old savepoint will be deleted and a new savepoint with the existing name will be created.

FIBPlus uses the SetSavePoint: transaction component method:

```
SetSavePoint(<identifier>;
```

Use the following SQL command to roll back to a certain savepoint:

```
ROLLBACK [WORK] TO [SAVEPOINT] <identifier>;
```

FIBPlus uses the following SetSavePoint: transaction component method to roll back the savepoint:

```
RollBackToSavePoint(<identifier>;
```

To release the savepoint use SQL RELEASE SAVEPOINT:

```
RELEASE SAVEPOINT <identifier> [ONLY];
```

If you don't set the keyword ONLY, all the savepoints (starting from the defined one) will be released and rolled back.

FIBPlus uses the ReleaseSavePoint: transaction component method:

```
ReleaseSavePoint(<identifier>;
```

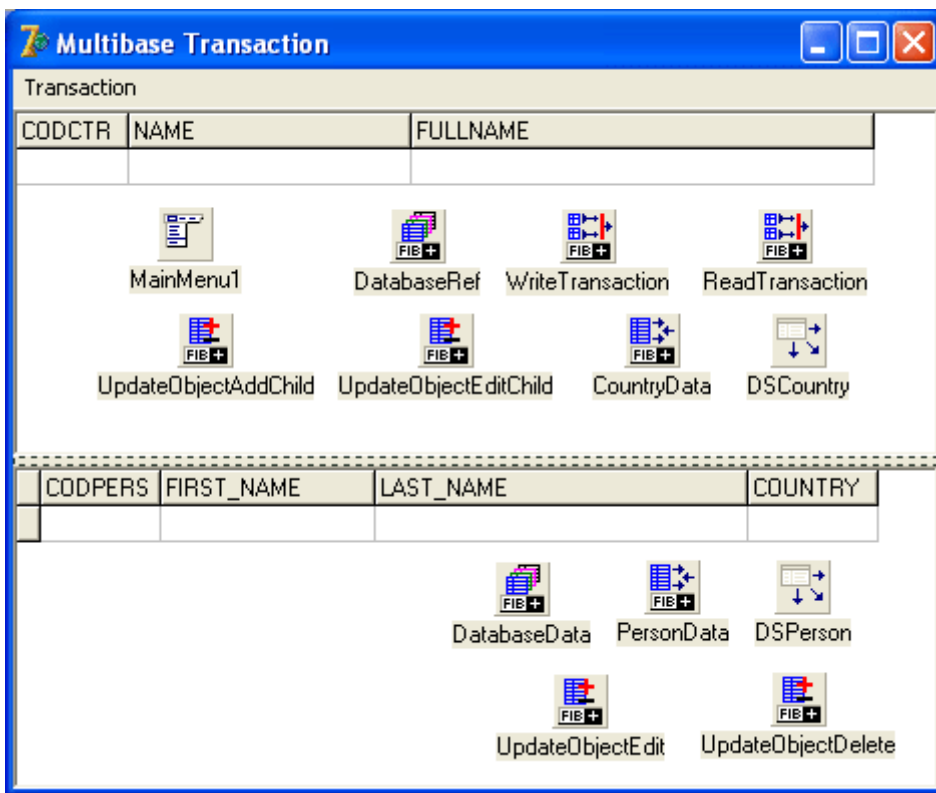
This method always releases all savepoints starting from the defined savepoint (this method is realized in version 6.25).

To demonstrate how to work with savepoints we will use the SavePoint example (with minor changes).

Create a new project. Place TPanel onto the form and align it to the right, it will be a toolbar. Then place a button closing the application, a pop-up ComboBox and five TButton's: to create a savepoint (see the text on the Add button), to roll back to a savepoint (Rollback), to commit the transaction (Commit), to release it (Release) and to add a savepoint with the existing name (AddExist).

Put StatusBar (to show a number of country records), DBGrid and DataSource. Add FIBPlus components to work with the database: TpFIBDatabase, TpFIBTransaction, TpFIBDataSet.

Now set SQL commands for the dataset, and define a long transaction for it (just do not set AutoCommit to True).



Picture 10. Project SavePoint

COUNTRY.GDB is a demo database.

The main issue in this project is how to write handlers. At first write the SavePoint:variable in private.

```
SavePoint: Integer;
```

There will be a current savepoint number. The OnShow event handler will be:

```
procedure TFormMain.FormShow(Sender: TObject);
begin
  Database.Connected := True;
  WriteTransaction.StartTransaction;
  CountryData.Open;
  SavePoint := 0;
  StatusBar1.Panels[1].Text := IntToStr(CountryData.RecordCount);
  DBGrid1.SetFocus;
end;
```

Just close the dataset and disconnect from the database after the application end. By default an active transaction will be rolled back. Now write the following click handler for savepoint saving:

```
procedure TFormMain.BSAddClick(Sender: TObject);
var NewPoint: string;
begin
  SavePoint := SavePoint + 1;
  NewPoint := 'SavePoint' + IntToStr(SavePoint);
  WriteTransaction.SetSavePoint(NewPoint);
  CSavePoints.Items.Add(NewPoint);
  CSavePoints.ItemIndex := CSavePoints.Items.Count - 1;
  DBGrid1.SetFocus;
end;
```

Here you create a savepoint name (it must be unique for the transaction execution context) and put it into ComboBox. The savepoint creation command is:

```
WriteTransaction.SetSavePoint(NewPoint);
```

The transaction can be rolled back to the savepoint selected from ComboBox by clicking the Rollback button:

```
procedure TFormMain.BSRollbackClick(Sender: TObject);  
var NewPoint: string;  
    i, NewIndex: Integer;  
begin  
    if CSavePoints.ItemIndex < 0 then exit;  
    NewIndex := CSavePoints.ItemIndex - 1;  
    NewPoint := CSavePoints.Items.Strings[CSavePoints.ItemIndex];  
    WriteTransaction.RollbackToSavePoint(NewPoint);  
    CountryData.FullRefresh;  
    for i := CSavePoints.Items.Count - 1 downto NewIndex + 1 do  
    CSavePoints.Items.Delete(i);  
    CSavePoints.ItemIndex := NewIndex;  
    if NewIndex = -1 then CSavePoints.Clear;  
    SavePoint := CSavePoints.Items.Count;  
    StatusBar1.Panels.Items[1].Text := IntToStr(CountryData.RecordCount);  
    DBGrid1.SetFocus;  
end;
```

The Rollback command is the following:

```
WriteTransaction.RollbackToSavePoint(NewPoint);
```

Other commands only put savepoint names in order.

After committing the transaction you need to put the savepoint list to the initial state:

```
procedure TFormMain.BSCommitClick(Sender: TObject);  
begin  
    WriteTransaction.CommitRetaining;  
    CountryData.FullRefresh;  
    CSavePoints.Items.Clear;  
    CSavePoints.ItemIndex := -1;  
    SavePoint := 0;  
    DBGrid1.SetFocus;  
end;
```

The command for the savepoint releasing is similar to the Rollback command:

```
procedure TFormMain.BSReleaseClick(Sender: TObject);  
var NewPoint: string;  
    i, NewIndex: Integer;  
begin  
    if CSavePoints.ItemIndex < 0 then exit;  
    NewIndex := CSavePoints.ItemIndex - 1;  
    NewPoint := CSavePoints.Items.Strings[CSavePoints.ItemIndex];  
    WriteTransaction.ReleaseSavePoint(NewPoint);  
    CountryData.FullRefresh;  
    for i := CSavePoints.Items.Count - 1 downto NewIndex + 1 do  
    CSavePoints.Items.Delete(i);  
    CSavePoints.ItemIndex := NewIndex;  
    if NewIndex = -1 then CSavePoints.Clear;  
    SavePoint := CSavePoints.Items.Count;  
    DBGrid1.SetFocus;  
end;
```

In ComboBox savepoints are placed according to the time of creation, so if you need to create a savepoint with the existing name you should delete the corresponding line from the list and add the savepoint name to the end of the list:

```
procedure TFormMain.BSAddExistClick(Sender: TObject);  
var NewPoint: string;  
begin  
  if CSavePoints.ItemIndex < 0 then exit;  
  NewPoint := CSavePoints.Items.Strings[CSavePoints.ItemIndex];  
  CSavePoints.Items.Delete(CSavePoints.ItemIndex);  
  WriteTransaction.SetSavePoint(NewPoint);  
  CSavePoints.Items.Add(NewPoint);  
  CSavePoints.ItemIndex := CSavePoints.Items.Count - 1;  
  DBGrid1.SetFocus;  
end;
```

And finally you should write the event handler for the line deleting to correct the number of countries shown in StatusBar:

```
procedure TFormMain.CountryDataAfterDelete(DataSet: TDataSet);  
begin  
  StatusBar1.Panels.Items[1].Text := IntToStr(CountryData.RecordCount);  
end;
```

Here you are, execute the application. Create as many savepoints as necessary, change and delete data, return to any savepoint, release savepoints, create savepoints with existing names. Everything works fine.

As a long updating transaction is used for user savepoints, use this transaction with the SNAPSHOT TABLE STABILITY isolation level exclusively.

Conclusion

In this article I have considered in detail how to use transactions in InterBase/Firebird databases and used main transaction characteristics by writing applications for data access using FIBPlus components.

The READ COMMITTED isolation level is the best for simultaneous multi-user work. The Update transactions must be short.

The SNAPSHOT isolation level is not convenient because of a high probability of lock exceptions.

If you need exclusive access to some database tables, you should use SNAPSHOT TABLE STABILITY. Only remember that on starting such a transaction and trying to open datasets you can get a lock exception in case some concurrent transaction made changes in the table without committing the transaction. If you use short Update transactions, the lock exception probability is small.

If you use separate transactions to read and write data, the reading transaction must be READ COMMITTED. The writing transaction can have any isolation level and additional characteristics depending on the task.

The most suitable variants for writing transactions are the WAIT mode together with NO RECORD_VERSION and NO WAIT together with RECORD_VERSION.

WAIT with NO RECORD_VERSION enable you to avoid the lock exception after committing the transaction or rolling it back (the exceptions will arise if the transaction tries to change a record deleted by another process). In case of using NO WAIT and RECORD_VERSION you will get the exception immediately and then decide on taking other actions.

As a rule it's not good to use NO RECORD_VERSION if you have many concurrent users. When they are actively changing data in the database you can hardly start this transaction. In case you really need to get the latest changes and data updates, then this mode suits you most of all.

If you correctly use two transactions in FIBPlus (for reading and writing data), you can considerably increase the efficiency of server resource use and almost decrease users' problems in case of locks.

Using FIBPlus protected editing mode you prevent users from changing or deleting a record which is being edited by another user.

InterBase/Firebird servers (and FIBPlus components) can start transactions for multiple databases. In this case principles of work with multiple databases do not differ much from work with one DB. It's only important to link these databases. For this purpose FIBPlus has a very handy feature: theUpdateObject component.

Nested transactions help to fragment a long updating transaction and roll back its part instead of the whole.

I have been mentioning lock probability very often. Maybe that's why you can think that it's bad and you should avoid it as much as possible. Of course it's not right, as everything depends on the task you have. You often need to lock other clients' changes in the database. In practice you can use both: hard locking means (SNAPSHOT TABLE STABILITY) and the protected editing mode. InterBase/Firebird and FIBPlus components have various means for every task requiring to working with data in the client-server application.