

# Working With TSQLQuery

From Free Pascal wiki

| [English \(en\)](#) | [español \(es\)](#) | [français \(fr\)](#) | [中文 \(中国大陆\) \(zh\\_CN\)](#) |

## Contents

- 1 General
- 2 Official documentation
- 3 Commonly used controls
- 4 Updating data
- 5 Cached Updates
- 6 Primary key fields
- 7 Controlling the update
- 8 Customizing the SQL in TSQLQuery
  - 8.1 SQL - Basic SQL customization
  - 8.2 InsertSQL, UpdateSQL and DeleteSQL - Basic use of parameters
  - 8.3 Parameters in TSQLQuery.SQL
    - 8.3.1 Select query example
    - 8.3.2 Insert query example
  - 8.4 Query with Format function
- 9 Running your own SQL and getting metadata
- 10 Troubleshooting
  - 10.1 Logging
  - 10.2 Poor performance
  - 10.3 Error messages
  - 10.4 Out of memory errors
  - 10.5 Dataset is read-only

## Database portal

References:

- General info
- Libraries
- Field types
- Controls
- FAQ
- SQL how-to
- Working With TSQLQuery
- In-memory database applications

Tutorials/practical articles:

- Overview
- 0 - Database set-up
- 1 - Getting started
- 2 - Editing
- 3 - Queries
- 4 - Data modules
- SQLdb Programming Reference


Databases

Advantage - MySQL - MSSQL -  
Postgres - Interbase - Firebird - Oracle -  
ODBC - Paradox - SQLite - dBASE -  
MS Access - Zeos

## General

TSQLQuery is an object that can embody a dataset coming from a database (RDBMS that uses SQL, such as Firebird, MS SQL Server, Oracle...). Using a SELECT SQL statement in the TSQLQuery's SQL property, you can determine what data is retrieved from the database into the dataset. When the dataset is changed by the program (or user), the changes can be submitted back to the database.

A TSQLQuery can also be used to directly modify data: if you specify the desired INSERT, UPDATE, DELETE etc SQL statement in the SQL property and call the ExecSQL method of the TSQLQuery, the query object will send the SQL to the database without retrieving any results.

Apart from its use in FPC, Lazarus also provides a component: TSQLQuery 

## Official documentation

See TSQLQuery documentation (<http://www.freepascal.org/docs-html/fcl/sqlldb/tsqlquery.html>)

A lot of context-sensitive documentation is now available in Lazarus. Unfortunately, TSQLQuery does not appear in the index of Lazarus 1.0 help. If you place your cursor on TSQLQuery methods and properties, try pressing F1 to see if that code is documented; e.g. this will work:

```
var
Q: TSQLQuery
...
Q.Open; //<--- place cursor on Open and press F1
```

## Commonly used controls

The dataset returned by TSQLQuery can conveniently be *viewed* with an instance of TDBGrid, but it is not very suitable for *editing* the data in the individual fields and cells. For this purpose you need to place some Data-Aware single-field controls such as TDBEdit on your form, and set their DataSource property to the data source being used. The DataField property should be set to a named field (eg 'IDENTITY') or to some expression that returns a suitable string.

Addition of a TDBNavigator toolbar makes it very easy to navigate through the records, and to select records for editing. When a record is selected by the toolbar or by moving the mouse through the data grid, the data for the relevant row and column appear in the TDBEdit box and if the 'Edit' button is clicked, the contents in the Edit box can be modified. Clicking on the 'Post' button confirms the change, or clicking on the 'Cancel' button cancels the changes.

In general, the process is as follows:

1. Drop a TSQLQuery on a form/datamodule, and set the *Database*, *Transaction* and *SQL* properties.
2. Drop a TDataSource component, and set its *DataSet* property to the TSQLQuery instance.
3. Drop a TDBGrid on the form and set its *DataSource* property to the TDataSource instance.
4. Optionally, drop a TDBNavigator instance on the form, and set its *Datasource* property to the TDataSource instance.

After this, the *Active* property can be set to *True*, and it should be possible to see the data retrieved by the query. (provided both the *TSQLConnection* and *TSQLTransaction* components are active)

## Updating data

If you need to be able to DELETE or otherwise modify records, your database table should either

1. contain one PRIMARY KEY column.
2. have a set of fields that uniquely determine the record. Normally, they should be part of a unique index.  
This is not required, but will speed up the queries quite a lot

If there is no primary field, or no fields that uniquely determine your record, then a primary key field should be added. This is done preferably when the table structure is designed, at CREATE time, but can be added at a later time.

For instance the following example code in your MySQL client will add a unique index to your table:

```
alter table testrig
add column autoid int
primary key auto_increment;
```

Adding this field will not hurt, and will allow your applications to update the field.

## Cached Updates

The *TSQLQuery* component caches all updates. That is, the updates are not sent immediately to the database, but are kept in memory till the *ApplyUpdates* method is called. At that point, the updates will be transformed to SQL update statements, and will be applied to the database. If you do not call *ApplyUpdates*, the database will not be updated with the local changes.

## Primary key fields

When updating records, *TSQLQuery* needs to know which fields comprise the primary key that can be used to update the record, and which fields should be updated: based on that information, it constructs an SQL UPDATE, INSERT or DELETE command.

The construction of the SQL statement is controlled by the *UsePrimaryKeyAsKey* property and the *ProviderFlags* properties.

The *Providerflags* property is a set of 3 flags:

### **pfInkey**

The field is part of the primary key

### **pfInWhere**

The field should be used in the WHERE clause of SQL statements.

### **pfInUpdate**

Updates or inserts should include this field.

By default, *ProviderFlags* consists of *pfInUpdate* only.

If your table has a primary key (as described above) then you only need to set the *UsePrimaryKeyAsKey* property to *True* and everything will be done for you. This will set the *pfInKey* flag for the primary key fields.

If the table doesn't have a primary key index, but does have some fields that can be used to uniquely identify the record, then you can include the *pfInKey* option in the *ProviderFlags* property all the fields that uniquely determine the record.

The *UpdateMode* property will then determine which fields exactly will be used in the WHERE clause:

### **upWhereKeyOnly**

When *TSQLQuery* needs to construct a WHERE clause for the update, it will collect all fields that have the *pfInKey* flag in their *ProviderFlags* property set, and will use the values to construct a WHERE clause which uniquely determines the record to update -- normally this is only needed for an UPDATE or DELETE statement.

### upWhereChanged

In addition to the fields that have *pfInKey* in the *ProviderFlags* property, all fields that have *pfInWhere* in their *ProviderFlags* and that have changed, will also be included in the WHERE clause.

### upWhereAll

In addition to the fields that have *pfInKey* in the *ProviderFlags* property, all fields that have *pfInWhere* in their *ProviderFlags*, will also be included in the WHERE clause.

## Controlling the update

It is possible to specify which fields should be updated: As mentioned above: Only fields that have *pfInUpdate* in their *ProviderOptions* property will be included in the SQL UPDATE or INSERT statements. By default, *pfInUpdate* is always included in the *ProviderOptions* property.

## Customizing the SQL in TSQLQuery

Normally TSQLQuery will use generic SQL statements based on properties as discussed above. However, the generic SQL created by sqldb may not be correct for your situation. TSQLQuery allows you to customize SQL statements used for the various actions, to work best in your situation with your database. For this purpose you use the properties *SQL*, *InsertSQL*, *UpdateSQL* and *DeleteSQL*.

All these properties are of type TStringList, a list of strings, that accepts multiple lines of SQL. All four come with a property editor in the IDE. In the IDE, select the property and open the editor by clicking the ellipsis button. In this editor (TSQLQuery metadata tool), you may also look up table information etc.

In code, use for example `InsertSQL.Text` or `InsertSQL.Add()` to set or add lines of SQL statements. One statement may span several lines and ends with a semicolon.

Also, all four properties accept parameters explained further below.

### SQL - Basic SQL customization

The SQL property is normally used to fetch the data from the database. The generic SQL for this property is `SELECT * FROM fpdev where fpdev` is the table as set in the database.

The dataset returned by the generic SQL statement will be kind of rough. If you show the result in a TDBGrid, the order of the records may seem random, the order of the columns may not be what you want and the field names may be technically correct but not user friendly. Using customized SQL you can improve this. For a table called `fpdev` with columns `id`, `UserName` and `InstEmail`, you could do something like this:

```
SELECT id AS 'ID', UserName AS 'User', InstEmail AS 'e-mail' FROM fpdev ORDER BY id;
```

The dataset that results from the above query uses the field names as given in the query (`ID`, `User` and `e-mail`), the column order as given in the query and the records are sorted by their `id`.

### InsertSQL, UpdateSQL and DeleteSQL - Basic use of parameters

When you assign a SELECT query to an SQLQuery's **SQL** property the SQLQuery knows how to get data from the database. However, when using databound controls such as a DBGrid, SQLQuery will also need to be able

to insert, update and delete rows from the database based on the user's actions.

In order to speed development, SQLQuery can try and deduce the required SQL statements. If the SQL property exists and the **ParseSQL** property is true (which it is by default), SQLQuery will try to generate these statements by parsing the **SQL** property. SQLDB stores these statements in the **InsertSQL**, **UpdateSQL** and **DeleteSQL** properties.

However, sometimes the generated statements will not work (e.g. when inserting in tables with auto-increment/autonumber primary keys) or will be very slow. If needed, you can manually assign the statements.

The statements in the *InsertSQL*, *UpdateSQL* and *DeleteSQL* properties accept parameters that represent fields in the dataset. The following rules apply:

- Parameter names must be exactly the same as the field names used in the dataset. The field names in the dataset may be different from the column names in the table, depending on the used select statement (see above).
- Just as parameters in other SQLDB queries, parameter names must be written preceded by a colon.
- For use in update/delete statements, precede the dataset field name with **OLD\_** (**strictly uppercase**, at least in Lazarus v. 1.0) to get the value of the record before it was edited instead of the new value.

If you have a table called `fpdev` and columns `id`, `UserName` and `InstEmail`, linked to a dataset with fields `ID`, `User` and `e-mail` (see example in select statement), you could write this **InsertSQL** query:

```
INSERT INTO fpdev(id, UserName, InstEmail) VALUES (:ID, :User, :e-mail);
```

This statement will insert the values of `ID`, `User` and `e-mail` from the current record of the dataset into the respective fields of table `fpdev`.

This example statement is actually more or less what SQLDB itself would autogenerate. The given statement may result in errors when the `id` field is an auto-increment field in a unique key. Different databases solve this problem in different ways. For example, the following works for MySQL.

```
INSERT INTO fpdev(id, UserName, InstEmail) VALUES (0, :User, :e-mail)
ON DUPLICATE KEY UPDATE id=LAST_INSERT_ID();
```

The above statement tries to insert a new record using 0 (zero) for the `id` column. If zero is already used as a key, then a duplicate is detected and the `id` is updated to use the last inserted id. Well, actually an id one increment higher than the last one used.

For Firebird, if you emulate autoincrement keys [1] (<http://www.firebirdfaq.org/faq29/>) something like this should work:

```
INSERT INTO fpdev(UserName, InstEmail) VALUES (:User, :e-mail); (
```

The statement inserts everything except the primary key and lets the Firebird before insert trigger use a generator/sequence to insert an `id` value for you.

For an `INSERT` statement you may want to use the current field values of the selected record. For `UPDATE` statements, you will want to use the field values as they were before editing in the `WHERE` clause. As mentioned before, the field values before editing must be written as the field name precede by **OLD\_** (**strictly uppercase**,

at least in Lazarus v. 1.0). For example, this query:

```
UPDATE fpdev SET UserName=:User, InstEmail=:e-mail WHERE UserName=:OLD_User;
```

The above statement updates the `UserName` and `InstEmail` columns of all records where `User` equals the old `User` value.

We leave it as an exercise to the reader to use the current field values and the old field values in `DELETE` statements.

See also the official documentation:

- `InsertSQL` documentation (<http://www.freepascal.org/docs-html/fcl/sqlldb/tsqlquery.insertsql.html>)
- `UpdateSQL` documentation (<http://www.freepascal.org/docs-html/fcl/sqlldb/tsqlquery.updatehtml.html>)
- `DeleteSQL` documentation (<http://www.freepascal.org/docs-html/fcl/sqlldb/tsqlquery.deletehtml.html>)

## Parameters in TSQLQuery.SQL

In most situations, the `SQL` property of `TSQLQuery` will contain the select statement which in most situations doesn't need parameters. However, it can contain them. This allows a very easy and powerful way to filter your records.

Parameters have the following advantages:

- no need to format your data as SQL text, date etc arguments (i.e. no need to remember how to format a date for MySQL, which might differ from the Firebird implementation; no need to escape text data like *O'Malley's "SQL Horror"*)
- possibly increased performance
- protection against SQL injection attacks

The use of parameters may help performance of the database. Most databases support prepared statements, which means that the statement is prepared and cached in the database. A prepared statement can be used more than once and doesn't require parsing and query planning every time it is used, only the parameters are changed each time it is used. In situations where the same statement is used a large number of times (where only the values of the parameters differ), prepared statements can help performance considerably. Additionally, SQL injection attacks can be mitigated by use of parameters.

The `InsertSQL`, `UpdateSQL` and `DeleteSQL` properties have predefined parameters for current and old field values. However, the `SQL` property does not. You can create your own parameters in the `Params` property.

## Select query example

This example shows how to select data using parameters. It also demonstrates using aliases (... AS ...) in SQL.

```
sql_temp.sql.text := 'SELECT id AS ''ID'', UserName AS ''User'', InstEmail AS ''e-mail'' FROM fpdev WHERE  
InstEmail=:emailsearch;'  
...  
//This will create a parameter called emailsearch.  
  
//If we want to, we can explicitly set what kind of parameter it is... which might only be necessary if FPC  
guesses wrong:
```

```
//sql_temp.params.parambyname('emailsearch').datatype:=ftWideString

//We can now fill in the parameter value:
sql_temp.params.parambyname('emailsearch').asstring := 'mvancanneyt@freepascal.org';
...
//Then use your regular way to retrieve data,
//optionally change the parameter value & run it again
```

## Insert query example

This example shows how to insert a new record into the table using parameters:

```
sql_temp.sql.text := 'insert into PRODUCTS (ITEMNR,DESCRIPTION) values (:OURITEMNR,:OURDESCRIPTION) '
...
sql_temp.Params.ParamByName('OURITEMNR').AsString := 'XXXX';
sql_temp.Params.ParamByName('OURDESCRIPTION').AsString := 'description';
sql_temp.ExecSQL;
SQLTransaction1.Commit; //or possibly CommitRetaining, depending on how your application is set up
```

Another way of doing this is something like:

```
tsqlquery1.appendrecord(['XXXX', 'description'])
tsqltransaction1.commit; //or commitretaining
```

## Query with Format function

Using parameterized queries is the preferred approach, but in some situations the format function can be an alternative. (see warning below). For example, parameters can't be used when you execute statements with the connection's ExecuteDirect procedure (*of course, you can just as well use a query to run the SQL statement in question*). Then this can come in handy:

```
procedure InsertRecord
var
  aSQLText: string;
  aSQLCommand: string;
begin
  aSQLText:= 'INSERT INTO products(item_no, description) VALUES(%d, %s)';
  aSQLCommand:= Format(aSQLText, [strtoint(Edit1.Text), Edit2.Text]);
  aConnection.ExecuteDirect(aSQLCommand);
  aTransaction.Commit;
end;
```

The values of the variables can change and the query values will change with them, just as with parameterized queries.

The parameter %d is used for integers, %s for strings; etc. See the documentation on the Format function for details.



**Warning:** Be aware that you may run into issues with text containing ' and dates using this technique!

## Running your own SQL and getting metadata

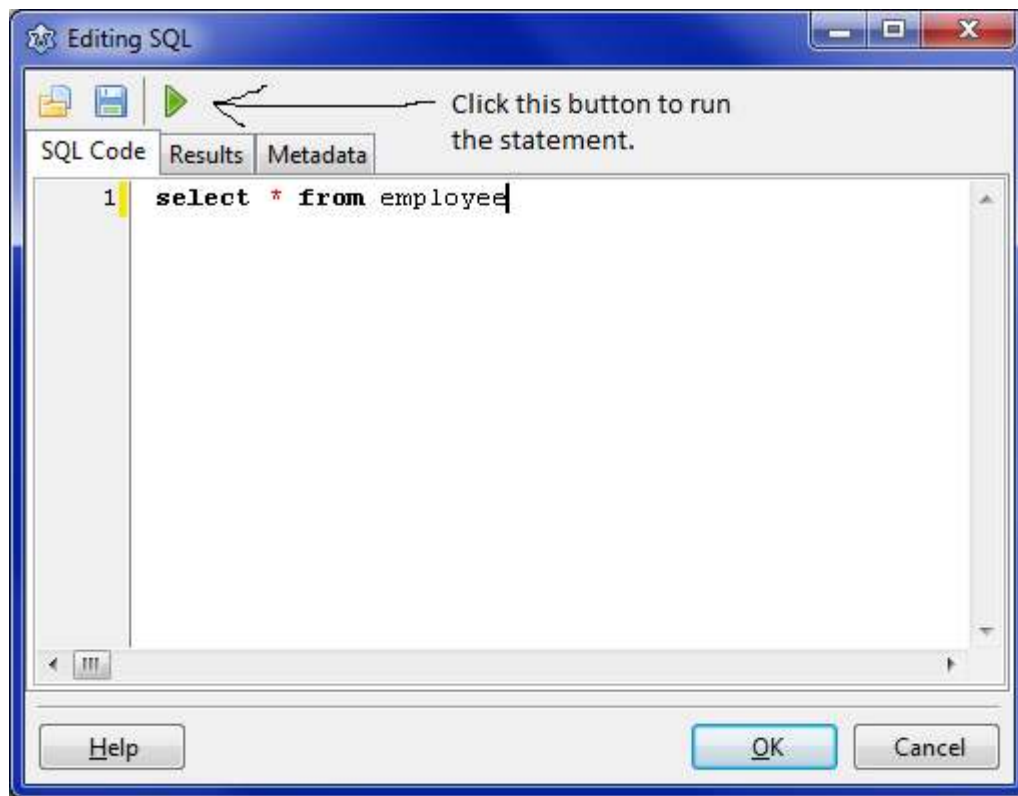
If you want to just check some SQL statements, troubleshoot, or get metadata (e.g. list of tables) from the

database, you can do so from within the IDE. In your program, with your T\*Connection, transaction, query object etc set up at design-time, go into the SQL property for the query object, then click the ... button.

You'll see a window with SQL code, and you can run some statements like

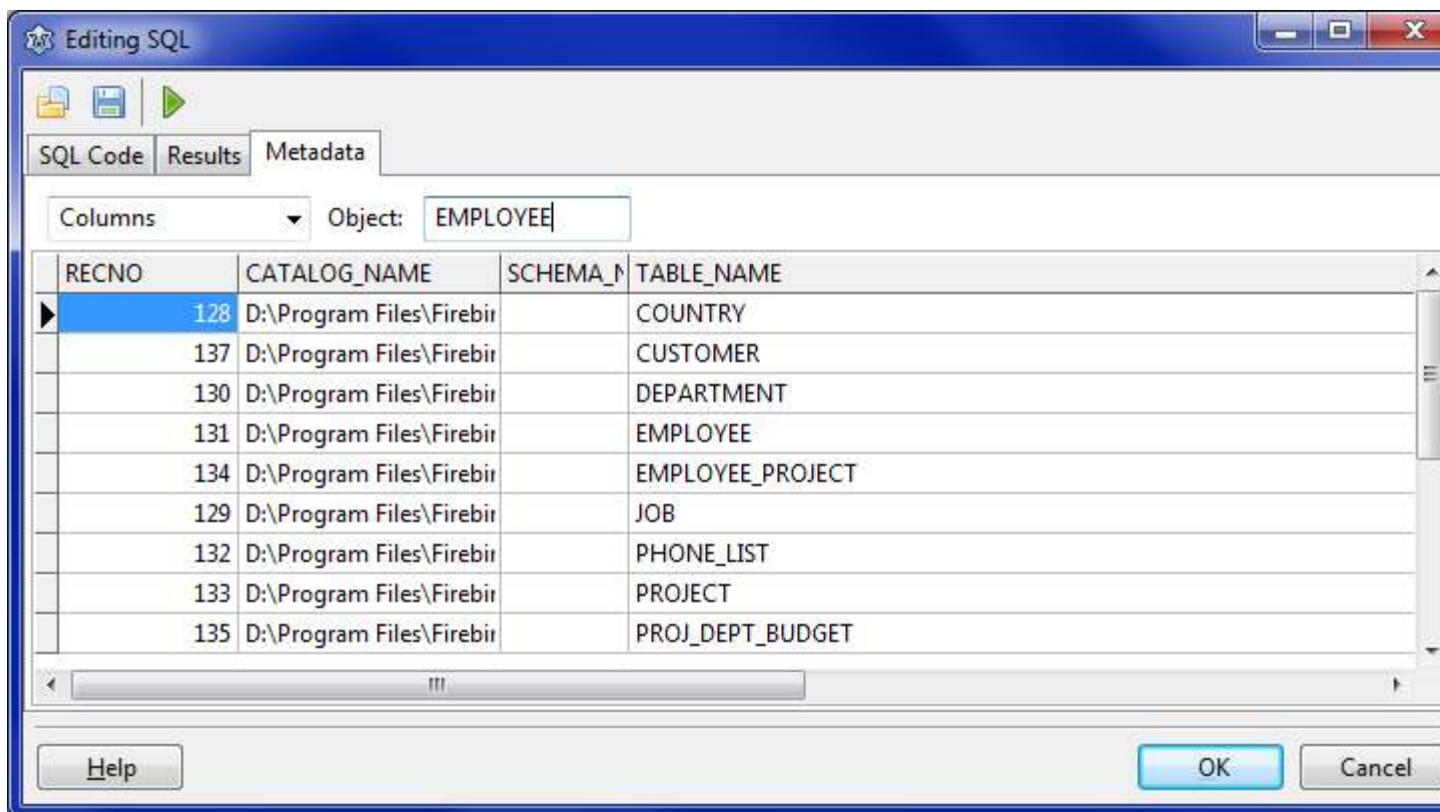
```
SELECT * FROM EMPLOYEE
```

by pressing the play icon:



You can also get metadata: table names, column names etc (if the sqldb connector supports it but most of them do nowadays):





(See also: Database metadata#Lazarus TSQLQuery metadata tool)

## Troubleshooting

### Logging

See here: [SqlDBHowto#Troubleshooting: TSQLConnection logging](#) for more detail.

### Poor performance

- Make sure your database queries are optimized (use proper indexes etc). Use your database tools (e.g. providing query plans) for this.
- See [#Out of memory errors](#) below for possible performance improvements when moving forward through an SQLQuery.

### Error messages

### Out of memory errors

TSQLQuery is a descendant of BufDataset, a dataset that buffers the data it receives into memory. If you retrieve a lot of records (e.g. when looping through them for an export), your heap memory may become full (with records you already looped through) and you will get out of memory errors.

Although this situation has improved in the FPC development version, a workaround is to tell bufdataset to

discard the records you have already read by setting the Unidirectional property to true before opening the query:

```
MySQLQuery.UniDirectional:=True;
```

This may also improve performance.

## Dataset is read-only

This may happen if you specify a query that you know is updatable but FPC doesn't.

Example:

```
select p.dob, p.surname, p.sex from people p;
```

The SQL parser in FPC is quite simplistic and when it finds a comma or space in the FROM part it considers multiple tables are involved and sets the dataset to read only. To its defense, aliasing tables is usually not done when only one table is involved. Solution: rewrite the query or specify your own InsertSQL, UpdateSQL and DeleteSQL.

Retrieved from "[https://wiki.freepascal.org/index.php?title=Working\\_With\\_TSQLQuery&oldid=107473](https://wiki.freepascal.org/index.php?title=Working_With_TSQLQuery&oldid=107473)"

- 
- This page was last edited on 24 February 2017, at 20:29.
  - Content is available under unless otherwise noted.