Overview

TMS Query Studio provides an easy way to give users access to powerful queries without requiring any knowledge about SQL. Users can setup complex queries in an almost natural language way with Query Studio. Dropping the component VisualQuery on the form and connect to the database opens the visual query power of Query Studio.

Query Studio offers two main components:

- TatVisualQuery component, which is a visual component with a friendly interface for end-users to build their own SQL statements.
- TatMetaSQL component, which is a non-visual component that encapsulates an SQL statement in a object-oriented architecture.

Feature details

- Visual definition of SQL, in a natural language-like way;
- Allows definition of source tables and joins;
- Allows definition of order and filtering;
- Exclusive parameter editors feature: a value in the filter condition can be attached to a parameter editor. End-user can change editor value and it will automatically be reflected in SQL;
- Supports SQL syntax for Microsoft Access, Microsoft SQL Server, Oracle, MySQL, Nexus, Interbase, DBISAM and Local BDE;
- Automatically set TDataset properties. Supports BDE, ADO, IBX, DBISAM, Nexus, IBO, DirectSQL and DBExpress dataset descendants;
- Automatic grouping feature for aggregate functions usage;
- Supports custom field expressions and custom filter conditions.

Rebuilding Packages

If for any reason you want to rebuild source code, you should do it using the "Packages Rebuild Tool" utility that is installed. There is an icon for it in the Start Menu.

Just run the utility, select the Delphi versions you want the packages to be rebuilt for, and click "Install".

If you are using Delphi XE and up, you can also rebuild the packages manually by opening the dpk/dproj file in Delphi/Rad Studio IDE.

Do NOT manually recompile packages if you use Delphi 2010 or lower. In this case always use the rebuild tool.

Installing Database Adapters

By default Query Studio package installs only TatCustomDatabase, TADODatbase, TDBXDatabase and TatBDEDatabase (if supported) components. Query Studio also provides other database connection components, which must be manually included in package before installing, since not every Delphi environments have the specific components installed:

AnyDAC

For installing TatAnyDACDatabase in your environment, add {$QS}\source\drivers\anydac directory to the Delphi library path, where {$QS} is the root directory of Query Studio files, and include qsanydacreg.pas unit in Query Studio package.

DBISAM

For installing TatDBISAMDatabase in your environment, add {$QS}\source\drivers\dbisam directory to the Delphi library path, where {$QS} is the root directory of Query Studio files, and include qsdbisamreg.pas unit in Query Studio package.

IBO

For installing TatIBODatabase in your environment, add {$QS}\source\drivers\ibo directory to the Delphi library path, where {$QS} is the root directory of Query Studio files, and include qsiboreg.pas unit in Query Studio package.

IBX

For installing TatIBXDatabase in your environment, add {$QS}\source\drivers\ibx directory to the Delphi library path, where {$QS} is the root directory of Query Studio files, and include qsibxreg.pas unit in Query Studio package.

NexusDB

For installing TatNexusDatabase in your environment, add {$QS}\source\drivers\nexus directory to the Delphi library path, where {$QS} is the root directory of Query Studio files, and include qsnxreg.pas unit in Query Studio package.

SQLDirect

For installing TatSQLDirDatabase in your environment, add {$QS}\source\drivers\sqldir directory to the Delphi library path, where {$QS} is the root directory of Query Studio files, and include qssqldirreg.pas unit in Query Studio package.

In this section:

Components Overview

Here is a brief summary of the installed components.

TatVisualQuery Component

TatVisualQuery is a component for visual SQL building. By dropping it on a form and setting some properties, your end-user will be able to build their own SQL statements with a friendly interface.

TatVisualQuery interface is tree-view like, where the first top node contains the name of the query being built. Each node below this one is related to an SQL part:

- Source tables: allows user to choose which tables records will come from, and how the tables will be linked.
- Data fields: allows choosing of fields that will be included in Select clause of SQL.
- Filter conditions: allows filtering of records (Where clause).
- Grouping fields: specificies grouping for records (which fields will be used for grouping).
- Ordering fields: specifieds fields for ordering records.
- Parameter editors: provides a higher-level interface where programmer or end-users can define parameters to be linked to SQL (similar to %parameter syntax in Tquery component). This way end-user can just change parameters in order to change SQL.

For more information about TatVisualQuery properties and methods, see the component reference at online help from IDE.

Quick start

In order to have TatVisualQuery component running, you should follow these steps:

1. Drop a TatVisualQuery component on a form.

2. Drop one TatDatabase components available in Query Studio palette. Choose the one which matches the db engine you want to use. Options are TatBDEDatabase, TatADODatabase, TatIBXDatabase or TatDBXDatabase.

3. Set TatDatabase component properties accordingly. For example, if using TatBDEDatabase, set its DatabaseName property to the database name you want to connect to.

4. Set TatVisualQuery.Databased property to point to the TatDatabase component you have just added.

The fours steps above will make TatVisualQuery to work. You will be able to construct queries, but will be better if you could USE the built queries. The built SQL is available in MetaSQLDef property:

```
BuiltMetaSQL := VisualQuery1.MetaSQLDef.MetaSQL.SQL;
```

However, TatVisualQuery component can perform a higher level of interface and update automatically the dataset component you want to use:

5. Drop the dataset component you want to use (for example, TQuery for BDE engine, or TADOQuery for ADO engine).

6. Set TatVisualQuery.TargetDataset component to point to the dataset component.

7. Set TatVisualQuery.AutoQueryOpen to True.

That's all you need to get it running. TatVisualQuery will automatically update and reopen your TQuery as end-user changes its query definitions. If you attach a DBGrid to the dataset, you will see query results.

TatCustomDatabase Class

TatCustomDatabase is a special TatDatabase which does not connect to any database. Instead, it provides events for you to "simulate" a database connection and retrieve field and table names from those events. Whenever you want your enduser to build an SQL but not connected to a database, use TatCustomDatabase component to provide the available tables and fields for the end-user.

Component events:

```
TatRetrieveTableNameListEvent = procedure(const AList: TStrings) of object;
property OnRetrieveTableNameListEvent: TatRetrieveTableNameListEvent;
```

Use this event to provide the names of tables which can be selected from. Fill the AList parameter with the table names.

Example:

```
AList.Add('Customer');
AList.Add('Orders');

TatRetrieveFieldNameListEvent = procedure(const ATableName: string; const AList:
TStrings) of object;
property OnRetrieveFieldNameListEvent: TatRetrieveFieldNameListEvent;
```

Use this event to provide the names and types of fields which can be selected in the table specified by ATableName parameter. Fill the AList parameter with the field names. The field type must be typecasted to a TObject and inserted in the Objects property of the list.

Example:

```
if UpperCase(ATableName) = 'CUSTOMER' then
begin
  AList.AddObject('CustNo', TObject(Ord(ftInteger)));
  AList.AddObject('Company', TObject(Ord(ftString)));
  //And so on...
end;
```

Custom database connections

Query Studio already provides several database connection components, descending from TatCustomDatabase. But since not every Delphi environments have the specific components installed, you must install the components manually, by following instructions here.

TatMetaSQLDefs and TatMetaSQLDef classes

TatMetaSQLDefs class holds a collection of TatMetaSQLDef classes, which in turn contains information of queries built in TatVisualQuery. TatVisualQuery holds this collection in MetaSQLDefs property, and it is built this way (as a collection) because it can hold more that just one query. Once the TatVisualQuery component holds more than one query, end-user can choose what query to edit/execute by right - clicking on left top icon of TatVisualQuery component and choosing the query from the popup menu that is displayed.

For more information about TatMetaSQLDef properties and methods, see the component reference at online help from IDE.

Localization

TMS Query Studio provides an easy way to localize the strings. All strings used in user interface (messages, button captions, dialog texts, menu captions, etc.) are in a single file named qsLanguage.pas.

In the *languages* folder, included in Query Studio distribution, there are several qsLanguage.pas files available for different languages. Just pick the one you want and copy it to the official directory of your query studio source code.

If the language you want does not exist, you can translate it yourself. Just open qsLanguage.pas file and translate the strings to the language you want.

As a final alternative, you can translate the qsLanguage.txt file, also included in *languages* folder, and send the new file to us. The advantage of this approach is that this file is easier to translate (you don't have to deal with pascal language) and can be included in the official Query Studio distribution. This way we keep track of changes in translable strings and all new strings are marked in the upcoming releases. This way, you will always know what is missing to translate, and do not need to do some kind of file comparison in every release of Query Studio.

So, in summary, to localize Query Studio strings:

- **Option 1**
  - Pick the correct qsLanguage.pas file from any subfolder from the *languages* directory, according to the language you want.
  - Replace the official qsLanguage.pas (in source code directory) by the one you picked.
- **Option 2**
  - Translate the official qsLanguage.pas directly.
- **Option 3**
  - Translate the qsLanguage.txt file and send it to us (support@tmssoftware.com).
  - We will send you back a translated qsLanguage.pas file and this translation will be included in official release.

Using Parameters

One interesting feature in TatVisualQuery component is the usage of parameters. While building filtering conditions in TatVisualQuery, end-user follows the known procedures: add a condition, choose a field which will be compared to a value, choose operator (equal to, less than, etc.), and choose the value.

However, there is an option where the end-user do not compare a field to a value, but instead compare a field to a parameter. Instead of filling a specific value, end-user types the parameter name. Using parameter feature makes it easy to later change the value to be compared to a field: instead of going there to change the condition expression, user just change parameter value.

Whenever a new parameter name is typed in filter condition, user can then create the editor that will be used to edit the parameter value. This is done in the "Parameter editors" section of TatVisualQuery component. Definitions made in parameter editors are saved in the ParamDefs property of a TatMetaSQLDef class. The ParamDefs property is a TatParamDefs class, which holds a collection of TatParamDef classes. Each TatParamDef class contains specifications of a single parameter editor. Below is the reference of TatParamDef class.

For complete information about componentes related to query parameters, see the Query Studio component reference at online help from IDE.

TatParamDef Class

TatParamDef class holds information about a single parameter defined in the query. A query parameter can be one of the following types:

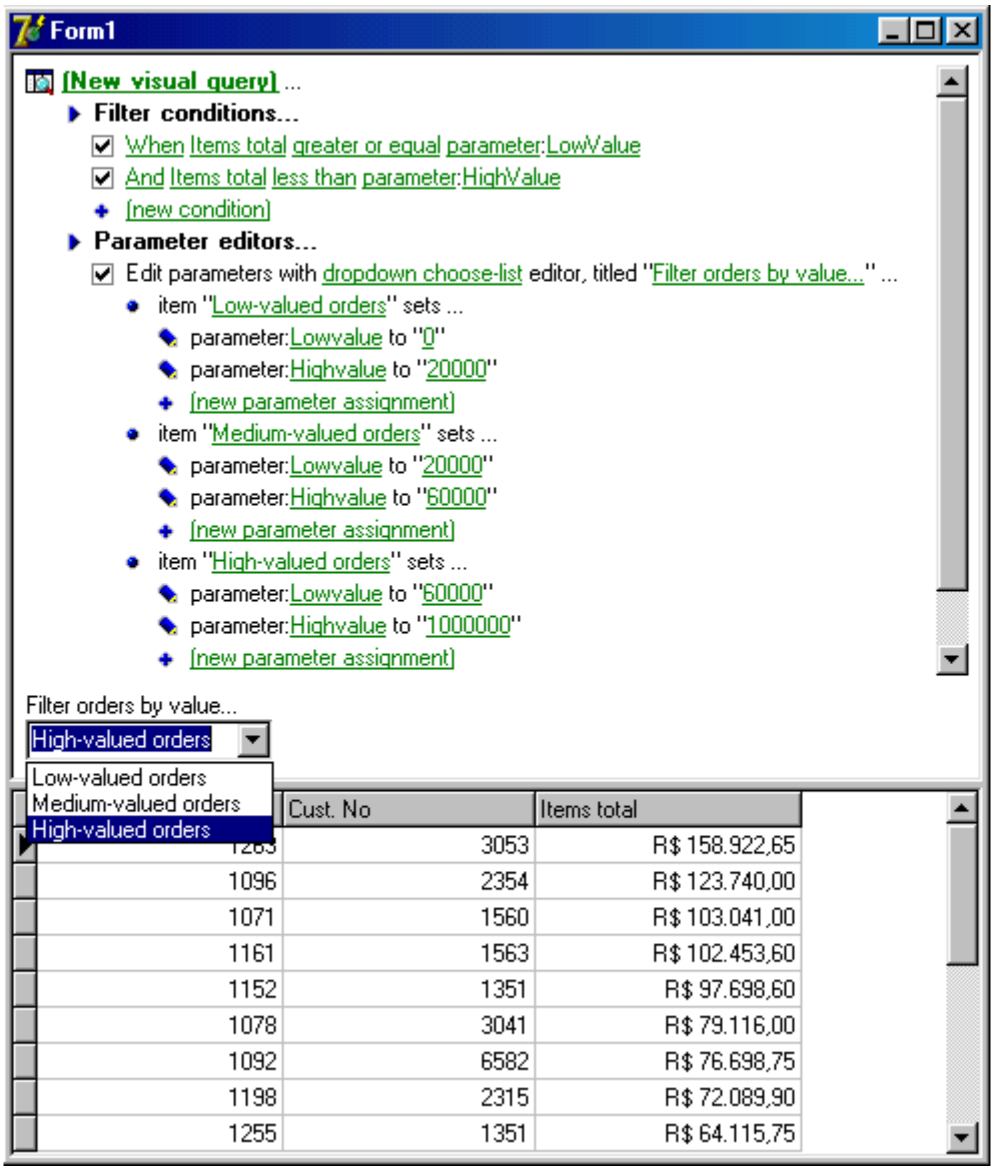
- ptFreeEdit: A simple edit component will be used to edit the parameter. User must type the parameter value.
- ptDistinctList: A combo box will be used to edit the parameter. User can type the parameter value or choose one option from combo items. The items in combo will be automatically filled with content of database. For example, if the parameter is "Customer city", and the field "CustomerCity" will be compared to this parameter, the combo items will be filled with all possible city names. A distinct query is made to the field in database to get all distinct values for the field.
- ptQueryList: A combo box will be used to edit the parameter. The items of the combo will be the query result of the SQL statement specified in MetaSQLStr property. The different between ptDistinctList and ptQueryList is that the query in distinct list is automatically built by TatVisualQuery, while with ptQueryList it's the user that specified the SQL statement.
- ptChooseList: A combo box will be displayed for editing the parameter. Each item in combo box is related to an item in ListItems property. Once the user chooses the combo box item, the related ListItem is applied. See ListItems property for more details.
- ptCheckList: Same as ptChooseList, but a check combo box is displayed instead. User can then choose one or more items, so that more than one ListItem can be applied.

For more information about TatParamDef properties and methods, see the component reference at online help from IDE.

Using choose-list and check-list parameters

TatParamListItem content is "applied" to SQL whenever user choose to apply it, by selecting/ checking an item in the parameter combo box.

What TatParamListItem contains is information about which parameters of sql will be changed and their values. This is done by using property ParamValues and its properties Values and Names. The screenshot below illustrates how to use choose list and check list parameters.



[New visual query] ...
- Filter conditions...
  - ☑ When Items total greater or equal parameter:LowValue
  - ☑ And Items total less than parameter:HighValue
  - [new condition]
- Parameter editors...
  - ☑ Edit parameters with dropdown choose-list editor, titled "Filter orders by value..." ...
    - item "Low-valued orders" sets ...
      - parameter:Lowvalue to "0"
      - parameter:Highvalue to "20000"
      - [new parameter assignment]
    - item "Medium-valued orders" sets ...
      - parameter:Lowvalue to "20000"
      - parameter:Highvalue to "60000"
      - [new parameter assignment]
    - item "High-valued orders" sets ...
      - parameter:Lowvalue to "60000"
      - parameter:Highvalue to "1000000"
      - [new parameter assignment]

Filter orders by value...

High-valued orders

- Low-valued orders
- Medium-valued orders
- High-valued orders

	Cust. No	Items total
1263	3053	R$ 158.922,65
1096	2354	R$ 123.740,00
1071	1560	R$ 103.041,00
1161	1563	R$ 102.453,60
1152	1351	R$ 97.698,60
1078	3041	R$ 79.116,00
1092	6582	R$ 76.698,75
1198	2315	R$ 72.089,90
1255	1351	R$ 64.115,75

Filter conditions are set to filter orders given by its ItemsTotal field, which value must be bettwen LowValue parameter and HighValue parameter. In "Parameter editors" section, a single parameter editor is created, of type "dropdown choose-list" (which is ptChooseList type). Three items were created for this parameter editor: "Low-valued orders", "Medium-valued orders" and "High-valued orders". Each of item correspond to a TatParamListItem object. For each list item, two parameter values were defined. For example, for the item name "Mediu-valued orders", the parameter "LowValue" receives 20000, and the parameter "HighValue" receives 60000.

After all is set, end-user will see a combo with a caption "Filter orders by value...", and the combo has three options: "Low-valued orders", "Medium-valued orders" and "High-valued orders". User just choose an item, and the correct orders are displayed in grid. This is the mechanism: let's say end-user chooses "Medium-valued orders" item. When this happen, the parameter values defined in "Medium-valued orders" item is set: LowValue parameter receives 20000, and HighValue parameter receives 60000. Since these two parameters are being used in "Filter conditions" section to filter records by ItemsTotal field, the query will bring all orders which ItemsTotal field value is between 20000 and 60000.

Finally, the TatParamListItem object reflects the parameter editor item above. Below is reference for TatParamListItem class, with examples related to the screenshot above.

TatMetaSQL Component

TatMetaSQL is a component that encapsulates an SQL statement, keeping in its properties all info needed to build the SQL statement, like fields to be selected, tables involved, order fields, and so on. The primary use for TatMetaSQL component is to allow building of SQL statements in an easy way. Here are some advantages of using it:

- There is no need to know SQL syntax – all that you need to do is to manipulate properties and objects inside TatMetaSQL;
- There is no need to care about target database – TatMetaSQL will build SQL statement for you, with correct syntax for Oracle, Microsoft SQL Server, Interbase, and more;
- It is very easy to have DB applications that works for multiple databases – thanks to feature described above;
- TatMetaSQL contains a visual design-time editor for SQL, allowing visual building of SQL statement;
- It is easy to allow end-user to change SQL – you can also use visual components that does that for you with few lines of code.

For complete information about TatMetaSQL methods and properties, see the Query Studio component reference at online help from IDE.

Working with TatMetaSQL – step by step

Basic example

This example ilustrates how to use TatMetaSQL. Let's say that you just want to query the customers database. Here is how you do in normal way:

```
MyQuery1.SQL.Text := 'Select C.CustNo from Customer C';
```

Here is how you do with TatMetaSQL:

```
with MetaSQL.SQLTables.Add do
begin
  TableName := 'Customer';
  TableAlias := 'C';
end;
with MetaSQL.SQLFields.Add do
begin
  FieldName := 'CustNo';
  TableAlias := 'C';
end;
MyQuery1.SQL.Text := MetaSQL.SQL;
```

Looks like it is more complicated, but suppose that now you want to change SQL in order to select both CustNo and Company fields. In the classic way, you would just change SQL text string. With MetaSQL, you do this:

```
with MetaSQL.SQLFields.Add do
begin
  FieldName:='Company';
  TableAlias:='C';
end;
```

Now imagine that you want to allow your end-user to do it. And not only add fields to select, but changing order, filtering, and so on. It will be very complicated to change SQL string, specially if the SQL is complex. TatMetaSQL makes it simpler.

Defining tables to query

From now, we are going to build an SQL step-by-step. First, set SQL syntax:

```
MetaSQL.SQLSyntax := ssBDELocal;
```

Now, first step to build an SQL statement is to define tables to query. This is done using TatMetaSQL.SQLTables property:

```
with MetaSQL.SQLTables.Add do
begin
  TableName:='Customer';
  TableAlias:='C';
end;
```

Which generated the SQL:

```
SELECT
  *
FROM
  Customer C
```

All references to a table in TatMetaSQL is done using the table alias (specified in TableAlias property). So, you must not add two tables with same TableAlias. Now if we add one more table:

```
with MetaSQL.SQLTables.Add do
begin
  TableName:='Orders';
  TableAlias:='O';
end;
```

TatMetaSQL will be aware that you will query two tables. But you must then indicate how these two tables are linked (see Defining table links).

Defining fields to be selected

The main property in TatMetaSQL component is the SQLFields property. Here you define not only fields to be selected, but also fields that can you can order by, group by or that you can use in conditions, inside where clauses or join clauses.

The code below add four fields in TatMetaSQL (CustNo and Company, from Customer table, and CustNo and OrderNo, from Orders table):

```
with MetaSQL.SQLFields.Add do
begin
  DataType:=ftFloat;
  FieldName:='CustNo';
  TableAlias:='C';
  FieldAlias:='Customer_CustNo';
end;
with MetaSQL.SQLFields.Add do
begin
  DataType:=ftString;
  FieldName:='Company';
  TableAlias:='C';
  FieldAlias:='Company';
end;
with MetaSQL.SQLFields.Add do
begin
  DataType:=ftFloat;
  FieldName:='CustNo';
  TableAlias:='O';
  FieldAlias:='Orders_CustNo';
end;
with MetaSQL.SQLFields.Add do
begin
  DataType:=ftFloat;
  FieldName:='OrderNo';
  TableAlias:='O';
  FieldAlias:='OrderNo';
end;
```

Here, TableAlias and FieldName property indicates the origin of field (from which table it comes, and the name of the field in database). And, FieldAlias property is the "ID" of the field in TatMetaSQL. Just like TableAlias for tables, FieldAlias is used by other parts of TatMetaSQL to make a reference to a specified field in SQLField property. Thus, you must not add to fields with same FieldAlias.

Defining table links

To define links (joins) between tables, you must use TatMetaSQL.TableJoins property:

```
with MetaSQL.TableJoins.Add do
begin
  PrimaryTableAlias:='C';
  ForeignTableAlias:='O';
  LinkType:=altInnerJoin;
  with JoinConditions.Add do
  begin
    ConditionType:=ctFieldCompare;
    FieldAlias:='Customer_CustNo';
    Operator:='=';
    Value:='Orders_CustNo';
  end;
end;
```

After this join, the resulting SQL is the following:

```
SELECT
  C.CustNo Customer_CustNo,
  C.Company Company,
  O.CustNo Orders_CustNo,
  O.OrderNo OrderNo
FROM
  Customer C,
  Orders O
WHERE
  ((
    C.CustNo = O.CustNo
  )
  )
```

To build the join, PrimaryTableAlias and ForeignTableAlias are used to indicate the table alias of tables that are to be linked. LinkType property is used to indicate if it will be inner join or outer join (left or right). After that, a condition is added to join, comparing the CustNo field of one table to another. Conditions will be described further.

Now let's show again the use of TatMetaSQL to easily change SQL. Suppose that we want to change the join for inner to outer (left join), in order to return all customers, even those that don't have a related record in Orders table:

```
MetaSQL.TableJoins.FindLink('C','O').LinkType := altLeftJoin;
```

This simple change will result in a different SQL:

```
SELECT
  C.CustNo Customer_CustNo,
  C.Company Company,
  O.CustNo Orders_CustNo,
  O.OrderNo OrderNo
FROM
  (Customer C LEFT JOIN Orders O
  ON (((
    C.CustNo = O.CustNo
  )
  )))
```

Defining order fields

In TatMetaSQL, you can define fields to order returned records. Use TatMetaSQL.OrderFields property to do that:

```
with MetaSQL.OrderFields.Add do
begin
  FieldAlias:='Customer_CustNo';
  SortType:=ortDescending;
end;
```

And result SQL is:

```
SELECT
  C.CustNo Customer_CustNo,
  C.Company Company,
  O.CustNo Orders_CustNo,
  O.OrderNo OrderNo
FROM
  (Customer C LEFT JOIN Orders O
  ON (((
    C.CustNo = O.CustNo
  )
  )))
ORDER BY
  C.CustNo DESC
```

Note that, one more time, field alias is used to tell meta sql which field the SQL should be ordered by. SortType property is used to indicate a descending order.

Defining conditions (filtering)

To finish our example SQL, we will use conditions to filter records. The code below will return only customers which number is less than 1250:

```
with MetaSQL.Conditions.Add do
begin
  ConditionType:=ctValueCompare;
  FieldAlias:='Customer_CustNo';
  Operator:='<';
  Value:=1250;
end;
```

Result SQL statement is:

```
SELECT
  C.CustNo Customer_CustNo,
  C.Company Company,
  O.CustNo Orders_CustNo,
  O.OrderNo OrderNo
FROM
  (Customer C LEFT JOIN Orders O
  ON (((
    C.CustNo = O.CustNo
  )
  )))
WHERE
  (
  C.CustNo < 1250
  )
ORDER BY
  C.CustNo DESC
```

A key property in a condition is ConditionType property. The value of this property will determine how condition expression will be built, based on properties FieldAlias, Operator, Value and Expression. Check the ConditionType property of Reference section of this document to know about the possible values of ConditionType.

Changing SQL syntax

After all tables and fields are defined, joins are built, order fields and conditions are set, what if we want to run this SQL on Oracle database? Just one line of code:

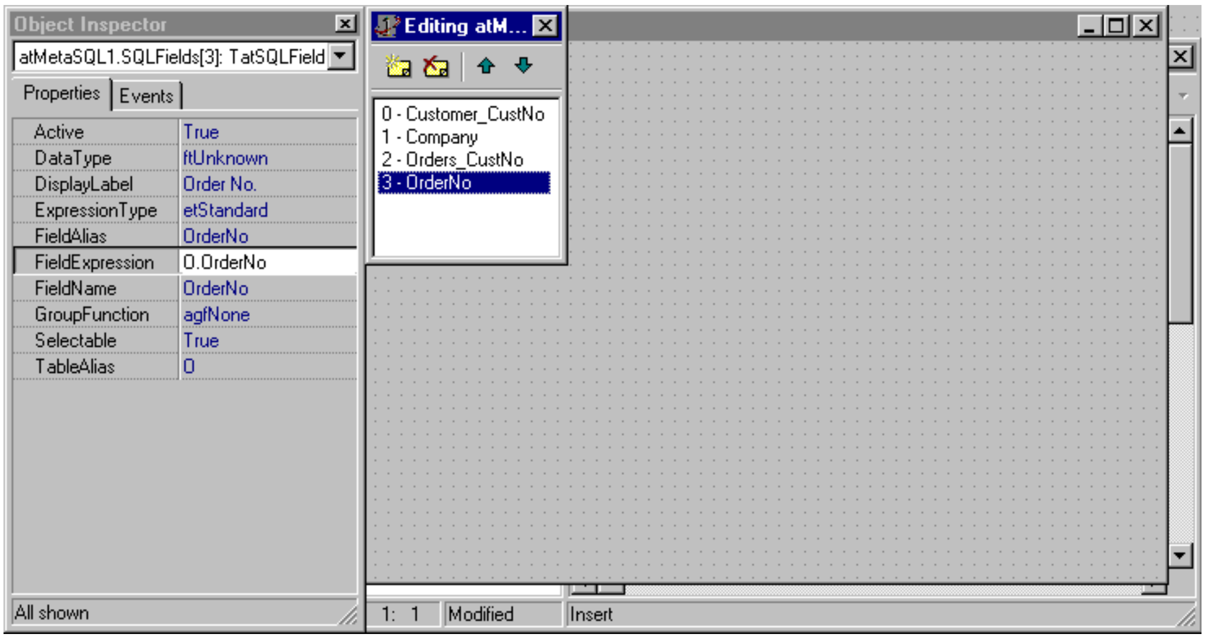
```
MetaSQL.SQLSyntax := ssOracle;
```

Now, the result SQL is:

```
SELECT
  C.CustNo Customer_CustNo,
  C.Company Company,
  O.CustNo Orders_CustNo,
  O.OrderNo OrderNo
FROM
  Customer C,
  Orders O
WHERE
  ((
    C.CustNo = O.CustNo(*)
  )
  )
AND
  (
  C.CustNo < 1250
  )
ORDER BY
  C.CustNo DESC
```

The MetaSQL visual editor

Previous chapter described main properties of SQL, and how to change it at runtime. But the most common way to build an SQL statement in TatMetaSQL is at design-time. All key properties like SQLFields, OrderFields, SQLTables, etc., can be manipulated using standard design-time Delphi object inspector. The figure below shows an example of editing/adding fields to be selected (SQLFields property).



atMetaSQL1.SQLFields[3]: TatSQLField

Active	True
DataType	ftUnknown
DisplayLabel	Order No.
ExpressionType	etStandard
FieldAlias	OrderNo
FieldExpression	O.OrderNo
FieldName	OrderNo
GroupFunction	agfNone
Selectable	True
TableAlias	O

0 - Customer_CustNo
1 - Company
2 - Orders_CustNo
3 - OrderNo

Design-time edit with standard Delphi object inspector

Another way to build SQL statement at design-time is using the TatMetaSQL editor. With visual editor, you can add/edit items like fields, tables and joins in an easy way. The following steps show how to use visual editor to build the same SQL built in previous chapter by code.

Visual editor overview

Visual editor has several tabs: Tables, Fields, Joins, etc., one for each important collection of items you can manipulate in MetaSQL. For each tab that you can add/edit/remove item, visual editor has a common part at the top: buttons New, Edit, Save and Cancel.

- New button: add a new item in list.
- Edit button: edit properties of selected item (you can also press Enter key when an item is selected).
- Save button: save changed properties in item (you can also press Enter key when editing item properties).
- Cancel button: cancel changes in item properties (you can also press Esc key when editing item properties).

To delete items, use Del key when item is selected.

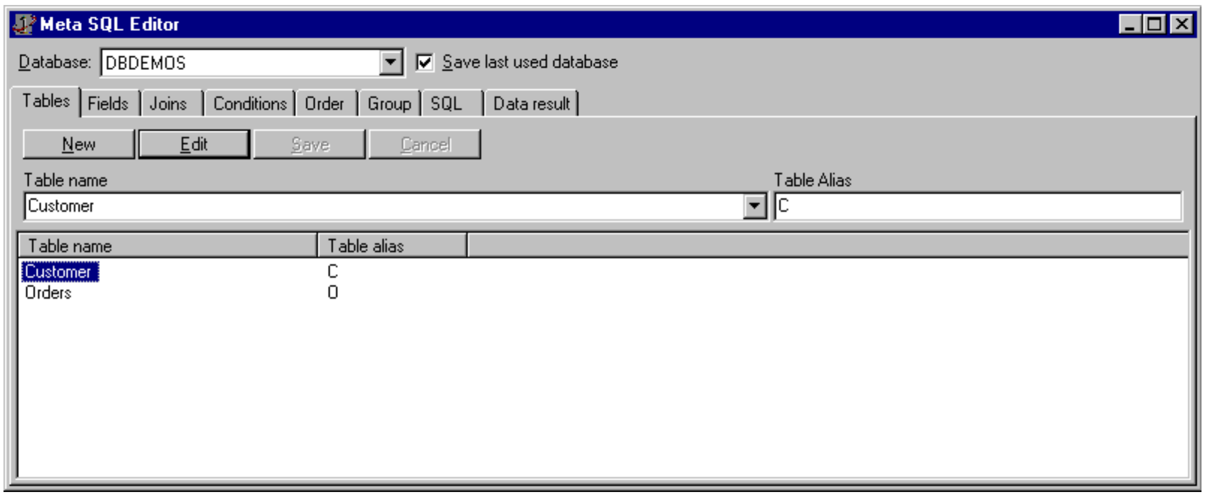
Choosing a database to work on

At the top of visual editor there is a combo box where you can choose a database alias. You don't need to choose a DB alias to use visual editor, but if you do it will help you. There are several places where you will need to choose a table name, a field name, a field type and so on, and if you have defined a DB alias, visual editor will show you a list of options (table names, field names, etc.) to choose from.

Checking option Save last used database will keep the selected DB alias next time you open visual editor.

Adding tables

Just like when building SQL by code, adding tables is the first step when using visual editor. Just use buttons to add tables, and set properties like TableName and TableAlias. The figure below shows the tables defined in visual editor.



Database: DBDEMOS

Save last used database [✓]

Table name: Customer

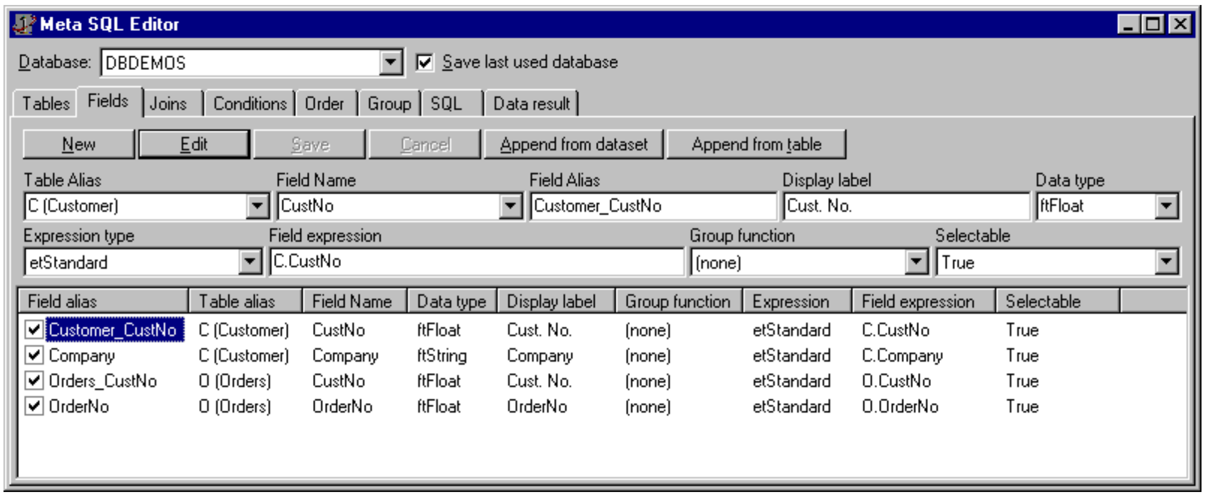
Table Alias: C

Table name	Table alias
Customer	C
Orders	O

Adding tables in visual editor

Adding fields

Next tab in visual editor is Fields, where you define fields to be selected and fields that will be used in conditions (filtering), ordering and grouping.



Database: DBDEMOS ☑ Save last used database

Table Alias: C (Customer) Field Name: CustNo Field Alias: Customer_CustNo Display label: Cust. No. Data type: ftFloat

Expression type: etStandard Field expression: C.CustNo Group function: (none) Selectable: True

Field alias	Table alias	Field Name	Data type	Display label	Group function	Expression	Field expression	Selectable
☑ Customer_CustNo	C (Customer)	CustNo	ftFloat	Cust. No.	(none)	etStandard	C.CustNo	True
☑ Company	C (Customer)	Company	ftString	Company	(none)	etStandard	C.Company	True
☑ Orders_CustNo	O (Orders)	CustNo	ftFloat	Cust. No.	(none)	etStandard	O.CustNo	True
☑ OrderNo	O (Orders)	OrderNo	ftFloat	OrderNo	(none)	etStandard	O.OrderNo	True

Including fields in visual editor

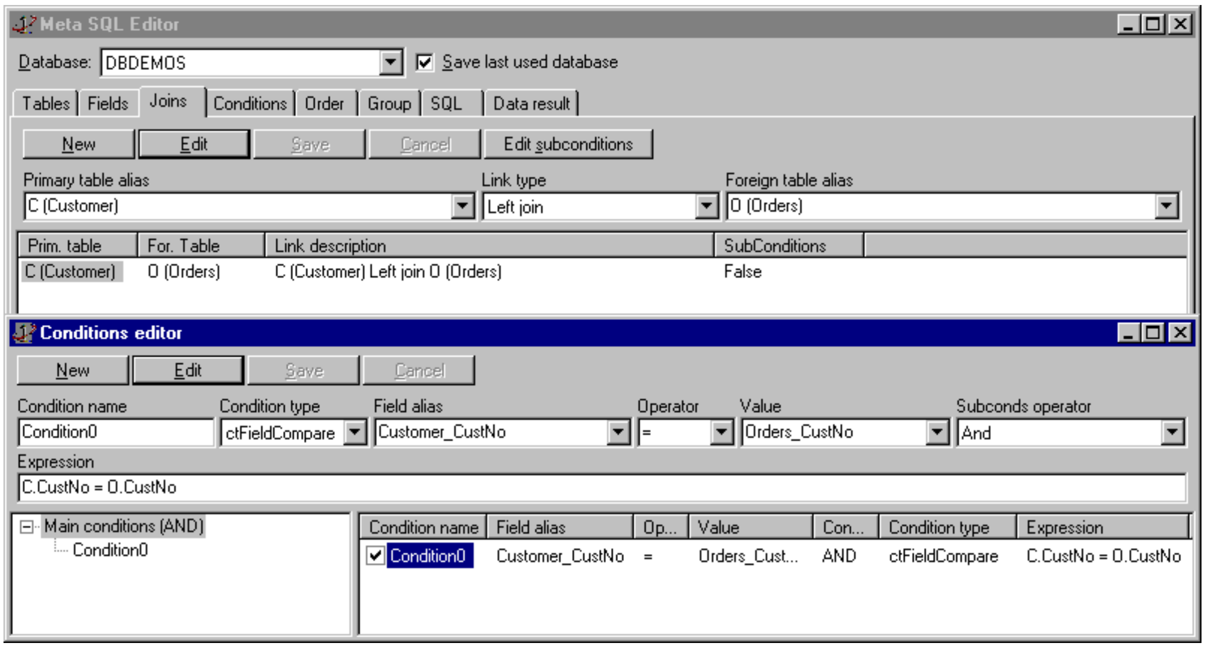
Note that here you can check/uncheck fields. This is because fields have an Active property. This checking/unchecking task will set Active property to true or false. To more info about Active property and the other field properties you can set here in visual editor, see the TatMetaSQL reference later in this document.

The Fields tab has also two extra buttons: Append from dataset and Append from table.

- Append from dataset button: Clicking this button will bring a list of TDataset descendant components (TTable, TQuery, etc.) that exists in the same form that own TatMetaSQL component. After choosing desired dataset component, the visual editor will create one new field in metasql for each persistent field in dataset, copying properties like FieldName, DisplayLabel and DataType.
- Append from table button: Only works if there is a database alias selected. Brings a list of existing tables in database. After choosing table, visual editor will create one new field in metasql for each existing field in database.

Defining table joins (links)

The Joins tab allow you to define table joins, just like by code. Here in the example of the figure below, only one join is added, between tables Customer and Orders.


Meta SQL Editor
Database: DBDEMOS
☑ Save last used database
Tables | Fields | Joins | Conditions | Order | Group | SQL | Data result
New | Edit | Save | Cancel | Edit subconditions
Primary table alias: C (Customer)
Link type: Left join
Foreign table alias: O (Orders)

Prim. table	For. Table	Link description	SubConditions
C (Customer)	O (Orders)	C (Customer) Left join O (Orders)	False

Conditions editor
New | Edit | Save | Cancel
Condition name: Condition0
Condition type: ctFieldCompare
Field alias: Customer_CustNo
Operator: =
Value: Orders_CustNo
Subconds operator: And
Expression: C.CustNo = O.CustNo
Main conditions (AND)
Condition0

Condition name	Field alias	Op...	Value	Con...	Condition type	Expression
☑ Condition0	Customer_CustNo	=	Orders_Cust...	AND	ctFieldCompare	C.CustNo = O.CustNo

Editing joins and join conditions in visual editor

In main windows of visual editor, you only add the join and set tables to join and link type. You must then add join conditions, use Edit subconditions button or double clicking the join. This will bring another window to edit join coniditions. The figure above shows both windows.

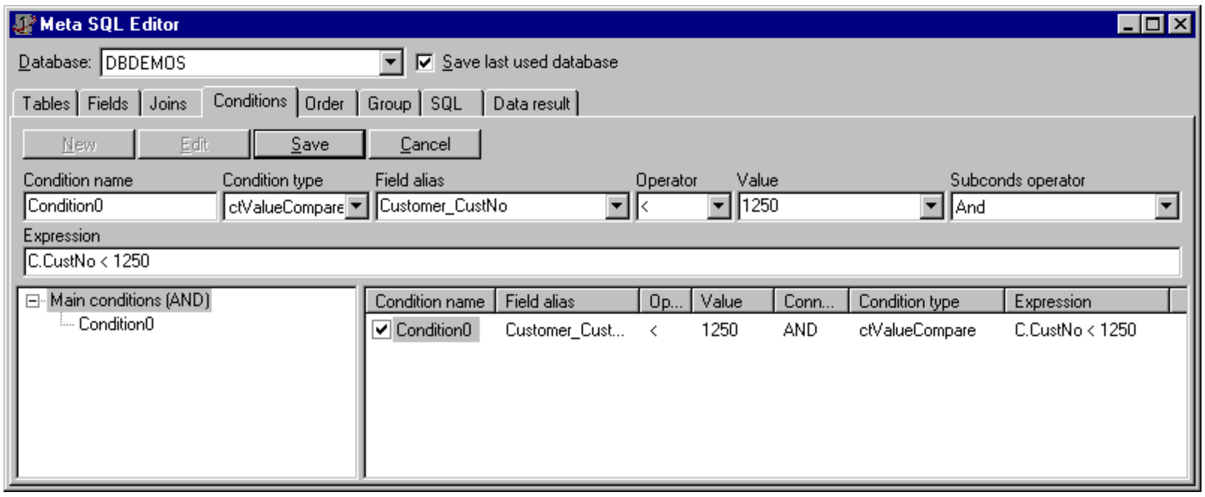
Each join must have a least one condition of type ctFieldCompare. In that condition you must define a comparison between fields of both tables being linked. Then you can add more conditions for the join, if you need to.

Defining conditions

In visual editor you can also define conditions, in the tab Conditions. To build the same SQL of previous chapter, we must define here a condition to filter SQL when customer number is less than 1250. The figure below shows the condition created.

Here the visual editor shows a different layout: there is a tree view at the left of condition list. This is because conditions are recursive: each condition has a property called SubConditions, which holds a collection of conditions. So, when a condition has subconditions, and its ConditionType property is set to ctSubConditions, than the condition is represented by its SubConditions, inside parentesis. This is useful to build nested conditions and group then with OR or AND logical operators.

In visual editor, the tree view at the left is used to see conditions structure. The root node Main conditions is not a condition itself, but represents the conditions that are directly create inside the TatMetaSQL.Conditions property. The "(AND)" after the name indicate how conditions inside that node will be grouped (in this case, with "and" operator).



Condition name: Condition0
Condition type: ctValueCompare
Field alias: Customer_CustNo
Operator: <
Value: 1250
Subconds operator: And
Expression: C.CustNo < 1250

Main conditions (AND)
Condition0

Condition name	Field alias	Op...	Value	Conn...	Condition type	Expression
✓ Condition0	Customer_Cust...	<	1250	AND	ctValueCompare	C.CustNo < 1250

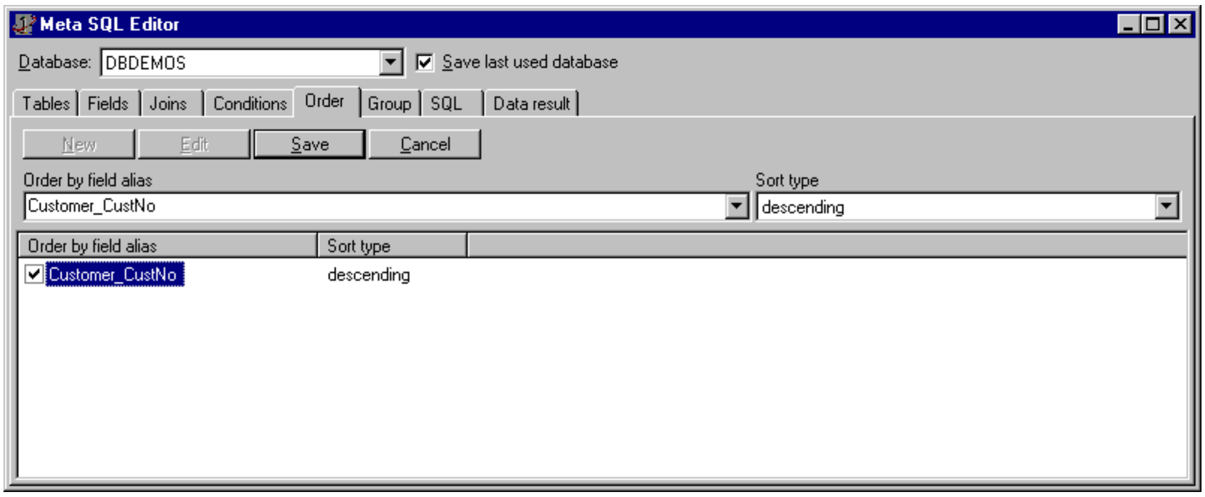
Adding conditions in visual editor

When you select a node in treeview, the condition list at the right will display the conditions of that node. In the example of figure above, the Main conditions node is selected, so all its conditions is displayed at the right (in this case, only condition Condition0). To add a new condition, you must select node first, and then click New button. This will create a new subcondition of selected condition in tree view.

Conditions also have Active property, and because of that there is a check box at the left of each condition. You can the activate or deactivate condition by checking/unchecking it. Non active conditions will not be included in SQL.

Defining order fields

Defining order fields in visual editor is simple: just add items and define Field alias and Sort type. The figure below shows the example, using the same order field and sort type of previou chapter (customer number, in descending order).



Order by field alias: Customer_CustNo
Sort type: descending

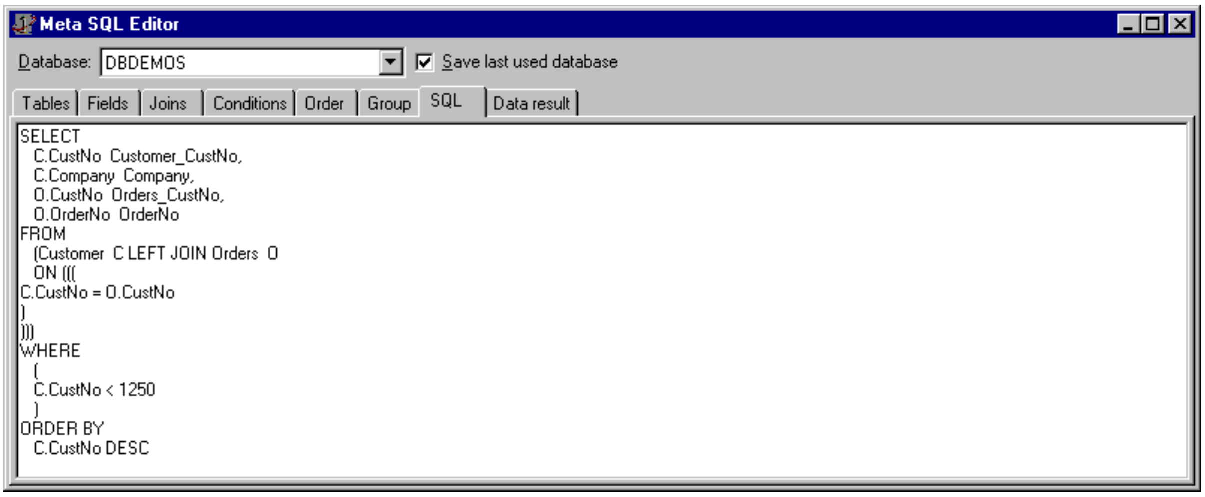
Order by field alias	Sort type
✓ Customer_CustNo	descending

Defining order fields

Order fields also have Active property. In visual editor, you use check box at the left of each order field to activate/deactivate orders. Only active order fields will be included in SQL.

Viewing SQL statement and data result

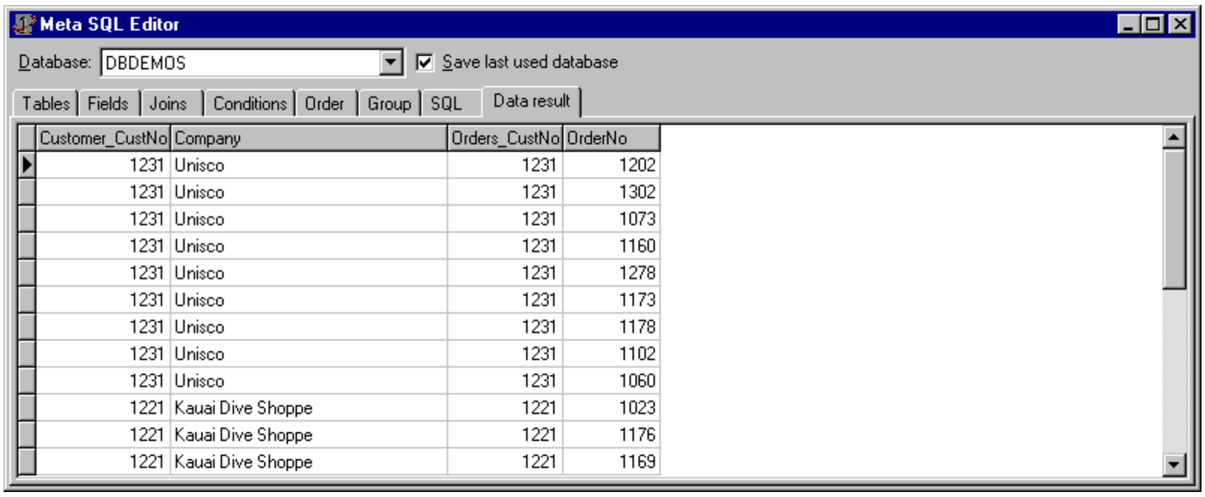
In visual editor, tab SQL displays the result SQL statement, as showed in the figure below. You cannot change SQL statement here, it is only for you to preview it and check if you have correctly built the Meta SQL.



```
SELECT
  C.CustNo  Customer_CustNo,
  C.Company  Company,
  O.CustNo  Orders_CustNo,
  O.OrderNo  OrderNo
FROM
  (Customer  C LEFT JOIN Orders  O
  ON (((
C.CustNo = O.CustNo
)
)))
WHERE
  (
  C.CustNo < 1250
  )
ORDER BY
  C.CustNo DESC
```

SQL statement preview

In addition to SQL statement preview, you can test the SQL in database, if you have defined a database alias. Tab Data result will execute the query in database and display a result dataset in a grid form. The figure below shows the data result for our SQL example.



Customer_CustNo	Company	Orders_CustNo	OrderNo
1231	Unisco	1231	1202
1231	Unisco	1231	1302
1231	Unisco	1231	1073
1231	Unisco	1231	1160
1231	Unisco	1231	1278
1231	Unisco	1231	1173
1231	Unisco	1231	1178
1231	Unisco	1231	1102
1231	Unisco	1231	1060
1221	Kauai Dive Shoppe	1221	1023
1221	Kauai Dive Shoppe	1221	1176
1221	Kauai Dive Shoppe	1221	1169

Result of execution of SQL statement in database

TatMetaSQL vs. SQL statement – quick comparison

For a quick reference, the tables below show the relation between a property in a class and its influence on SQL statement. The part of SQL statement that is manipulated by property is displayed in bold.

TatMetaSQL class

Property	**SQL Statement**
SQLTables	Select * from **Customer C, Order O, Parts P** where C.CustNo=2
SQLFields	Select **C.CustNo CustomerNumber, C.Company Company, O.OrderNo OrderNumber** from Customer C, Orders O where C.CustNo=O.CustNo
TableJoins	Select * from Customer **C Inner Join** Orders **O ON (C.CustNo=O.CustNo AND (C.CustNo=2 OR C.CustNo=3))**
Conditions	Select * from Customer C where **C.CustNo=10 or (C.Company Like 'A*' AND C.City IS NULL)**
GroupFields	Select O.CustNo, SUM(O.Price) from Orders O **Group By O.CustNo**
OrderFields	Select * from Customer C **Order By C.Country, C.City**
CustomFilter	Select * from Customer C where C.CustNo=2 AND **(C.City IN (Select P.City From City P where P.City like 'A*'))**
ConditionsLogicalOper	Select * from Customer C where C.CustNo=10 **OR** (C.Company Like 'A*' AND C.City IS NULL) **OR** C.CustNo=3

TatSQLField class

Property	**SQL Statement**
FieldAlias	Select C.CustNo **CustomerNumber** from Customer C
FieldName	Select C.**CustNo** CustomerNumber from Customer C
TableAlias	Select **C**.CustNo CustomerNumber from Customer C
GroupFunction	Select **Count(**C.CustNo**)** from Customer C
FieldExpression	Select **O.Price * O.Quantity** from Orders O

TatSQLOrderField class

Property	**SQL Statement**
FieldAlias	Select * from Orders O Order By **O.Date** Desc
SortType	Select * from Orders O Order By O.Date **Desc**

TatSQLGroupField class

Property	**SQL Statement**
FieldAlias	Select O.CustNo, SUM(O.Price) from Orders O Group By **O.CustNo**

TatSQLTable class

Property	**SQL Statement**
TableAlias	Select * from Customer **C**
TableName	Select * from **Customer** C

TatSQLCondition class

Property	**SQL Statement**
FieldAlias	Select * from Customer C where **C.Company** Like 'A*'
Operator	Select * from Customer C where C.Company **Like** 'A*'
Value	Select * from Customer C where C.Company Like **'A*'**
Expression	Select * from Orders O where **O.Price * O.Quantity > 2000**
SubConditionsLogicalOper	Select * from Customer C where C.CustNo=10 OR (C.Company Like 'A*' **AND** C.City IS NULL) OR C.CustNo=3
SubConditions	Select * from Customer C where C.CustNo=10 OR **(C.Company Like 'A*' AND C.City IS NULL)** OR C.CustNo=3

TatSQLJoin class

Property	**SQL Statement**
JoinConditionsLogicalOper	Select * from Customer C Inner Join Orders O ON (C.CustNo=O.CustNo **AND** (C.CustNo=2 OR C.CustNo=3))
JoinConditions	Select * from Customer C Inner Join Orders O ON **(C.CustNo=O.CustNo AND (C.CustNo=2 OR C.CustNo=3))**
PrimaryTableAlias	Select * from Customer **C** Inner Join Orders O ON (C.CustNo=O.CustNo AND (C.CustNo=2 OR C.CustNo=3))
ForeignTableAlias	Select * from Customer C Inner Join Orders **O** ON (C.CustNo=O.CustNo AND (C.CustNo=2 OR C.CustNo=3))
LinkType	Select * from Customer C **Inner Join** Orders O ON (C.CustNo=O.CustNo AND (C.CustNo=2 OR C.CustNo=3))

About

This documentation is for TMS Query Studio.

In this section:

What's New

Version 1.15 (Sep-2021)

- **New:** RAD Studio 11 support.

Version 1.14 (Jun-2020)

- New: RAD Studio 10.4 Sydney support.
- Fixed: Include files structure modified in 3rd party drivers to avoid hanging Delphi IDE in some situations.

Version 1.13 (Mar-2019)

- New: RAD Studio 10.3 Rio.
- New: TFireDacDatabase adapter available in folder source\drivers\firedac (fixed in 1.13.2).
- Fixed: Correct item height in TatVisualQuery component in High DPI mode (1.13.1).

Version 1.12 (Mar-2017)

- New: RAD Studio 10.2 Tokyo.
- Fixed: "Having" clause being wrongly included in SQL when using subconditions.

Version 1.11 (Jan-2017)

- New: Support for "is null" and "is not null" operators.
- Improved: Meta SQL generates HAVING clause allowing filtering by aggregated fields.

Version 1.10 (Apr-2016)

- New: RAD Studio 10.1 Berlin support.

Version 1.9.2 (Sep-2015)

- New: RAD Studio 10 Seattle support.

Version 1.9.1 (Apr-2015)

- New: Delphi/C++Builder XE8 support.

Version 1.9 (Mar-2015)

- New: Packages structure changed. Now it allows using runtime packages with 64-bit applications. It's a breaking change.

Previous Versions

Version 1.8.2 (Sep-2014)

- New: Delphi/C++ Builder XE6 support.
- New: Removed BDE dependency from the library.

Version 1.8.1 (May-2014)

- New: Delphi/C++ Builder XE6 support.

Version 1.8 (Oct-2013)

- New: Delphi/C++ Builder XE5 support.

Version 1.7 (May-2013)

- New: Delphi/C++ Builder XE4 support.

Version 1.6 (Sep-2012)

- New: Delphi/C++ Builder XE3 support.
- Fixed: Access Violation in MetaSQL editor.
- Fixed: Issue with lookup queries.

Version 1.5 (Sep-2011)

- New: Delphi/C++ Builder XE2 support.
- Fixed: Minor bug fixes.

Version 1.4.1 (Sep-2010)

- Fixed: Error compiling units of custom database connection components.
- Fixed: Compile errors related to unit Spin.pas.

Version 1.4 (Sep-2010)

- New: Localization support allows building queries using different languages than English.
- New: Support for AnyDAC components.

- New: Portuguese translation included.
- New: RAD Studio XE support.
- New: Help component reference.

Version 1.3

- New: ssFirebird syntax.
- New: SyntaxConf.IndexInGroupBy and SyntaxConf.IndexInOrderBy properties to allow syntax like "Order by 3, 5".
- New: SyntaxConf property which can be used to configure the syntax of SQL statement to be generated when SQLSyntax is set to ssCustom.
- New: SQL statement can also be retrieve from a TClientDataset when it is set to the TargetDataset of TatVisualQuery component.
- New: OnRetrieveTablenameListEvent and OnRetrieveFieldnameListEvent events available for all TatDatabase descendents.
- New: Fields in popup menu are now being displayed in alphabetical order.
- New: Events OnItemDeleting, OnItemDeleted, OnTreeViewParamChanged.
- New: Delphi 2006,2007 & C++Builder 2006,2007 support.
- New: CoInsertCustom option in TClauseOptions. Use this to turn allow/prohibit end-user to use the custom dialogs for building the query.

Version 1.2

- New: SQLDirect components support.
- New: SQL Parser now supports field and table names with spaces (e.g., Select "Customer Name" from "Customer Table").
- New: Property TatIBXDatabase.UseViews.
- New: Methods LoadQueriesFromStream and SaveQueriesToStream.
- New: IBO components support.
- Improved : Query Studio package does not longer require TMS Pack package when TMS Pack is installed
- Improved: Generated SQL statement does not duplicate order by fields, even if end-user duplicated it in visual query.
- Fixed: Disappearing check boxes after dragging fields.
- Fixed: Bug with "#sm#" string in table names.
- Fixed: "Could not parse SQL" with more than one space after "AND" and "OR" operators.

Version 1.0

- First release.

Copyright Notice

Getting Support

General notes

Before contacting support:

- Make sure to read the tips, faq and readme.txt or install.txt files in component distributions.
- Make sure you have the latest version of the component(s).

When contacting support:

- Specify with which component you have a problem.
- Specify which Delphi or C++Builder version you're using and preferably also on which OS.
- In case of IntraWeb or ASP.NET components, specify with which browser the issue occurs.
- For registered users, use the special priority support email address (mentioned in registration email) & provide your registration email & code. This will guarantee the fastest route to a solution.

Send email from an email account that

1. allows to receive replies sent from our server
2. allows to receive ZIP file attachements
3. has a properly specified & working reply address

Getting support

For general information: info@tmssoftware.com

Fax: +32-56-359696

For all questions, comments, problems and feature request for VCL components: help@tmssoftware.com

To improve efficiency and speed of help, refer to the version of Delphi, C++Builder, Visual Studio .NET you are using as well as the version of the component. In case of problems, always try to use the latest version available first.

Breaking Changes

List of changes in each version that breaks backward compatibility.

Version 1.9

- There was a big package restructuration in version 1.9, as detailed in the following topic.

Package Restructuration

TMS Query Studio packages have been restructured. The packages are now separated into runtime and design-time packages, allowing a better usage of them in an application using runtime packages (allows it to work with 64-bit applications using runtime packages, for example). Also, Libsuffix option is now being used so the dcp files are generated with the same name for all Delphi versions. Here is an overview of what's changed:

Before version 2.0, there was a single package named aquery<version>.dpk (where <version> is the "name" of delphi version), which generated BPL and DCP with same names:

Previous versions:

Version	Package File Name	BPL File Name	DCP File name
Delphi 7	aquery7.dpk	aquery7.bpl	aquery7.dcp
Delphi 2007	aquery2007.dpk	aquery2007.bpl	aquery2007.dcp
Delphi 2009	aquery2009.dpk	aquery2009.bpl	aquery2009.dcp
Delphi 2010	aquery2010.dpk	aquery2010.bpl	aquery2010.dcp
Delphi XE	aquery2011.dpk	aquery2011.bpl	aquery2011.dcp
Delphi XE2	aqueryxe2.dpk	aqueryxe2.bpl	aqueryxe2.dcp
Delphi XE3	aqueryxe3.dpk	aqueryxe3.bpl	aqueryxe3.dcp
Delphi XE4	aqueryxe4.dpk	aqueryxe4.bpl	aqueryxe4.dcp
Delphi XE5	aqueryxe5.dpk	aqueryxe5.bpl	aqueryxe5.dcp
Delphi XE6	aqueryxe6.dpk	aqueryxe6.bpl	aqueryxe6.dcp
Delphi XE7	aqueryxe7.dpk	aqueryxe7.bpl	aqueryxe7.dcp

From version 2.0 and on, there are two packages:

- TMSQueryStudio.dpk (runtime package)
- dclTMSQueryStudio.dpk (design-time packages)

DCP files are generated with same name, and only BPL files are generated with the suffix indicating the Delphi version. The suffix, however, is the same used by the IDE packages (numeric one indicating IDE version: 160, 170, etc.). The new package structure is as following (note that when 6.5 was released, latest Delphi version was XE7. Packages for newer versions will follow the same structure):

Version	Package File Name	BPL File Name	DCP File name
Delphi 7	TMSQueryStudio.dpk dclTMSQueryStudio.dpk	TMSQueryStudio70.bpl dclTMSQueryStudio70.bpl	TMSQueryStudio.dcp dclTMSQueryStudio.dcp
Delphi 2007	TMSQueryStudio.dpk dclTMSQueryStudio.dpk	TMSQueryStudio100.bpl dclTMSQueryStudio100.bpl	TMSQueryStudio.dcp dclTMSQueryStudio.dcp
Delphi 2009	TMSQueryStudio.dpk dclTMSQueryStudio.dpk	TMSQueryStudio120.bpl dclTMSQueryStudio120.bpl	TMSQueryStudio.dcp dclTMSQueryStudio.dcp
Delphi 2010	TMSQueryStudio.dpk dclTMSQueryStudio.dpk	TMSQueryStudio140.bpl dclTMSQueryStudio140.bpl	TMSQueryStudio.dcp dclTMSQueryStudio.dcp
Delphi XE	TMSQueryStudio.dpk dclTMSQueryStudio.dpk	TMSQueryStudio150.bpl dclTMSQueryStudio150.bpl	TMSQueryStudio.dcp dclTMSQueryStudio.dcp
Delphi XE2	TMSQueryStudio.dpk dclTMSQueryStudio.dpk	TMSQueryStudio160.bpl dclTMSQueryStudio160.bpl	TMSQueryStudio.dcp dclTMSQueryStudio.dcp
Delphi XE3	TMSQueryStudio.dpk dclTMSQueryStudio.dpk	TMSQueryStudio170.bpl dclTMSQueryStudio170.bpl	TMSQueryStudio.dcp dclTMSQueryStudio.dcp
Delphi XE4	TMSQueryStudio.dpk dclTMSQueryStudio.dpk	TMSQueryStudio180.bpl dclTMSQueryStudio180.bpl	TMSQueryStudio.dcp dclTMSQueryStudio.dcp
Delphi XE5	TMSQueryStudio.dpk dclTMSQueryStudio.dpk	TMSQueryStudio190.bpl dclTMSQueryStudio190.bpl	TMSQueryStudio.dcp dclTMSQueryStudio.dcp
Delphi XE6	TMSQueryStudio.dpk dclTMSQueryStudio.dpk	TMSQueryStudio200.bpl dclTMSQueryStudio200.bpl	TMSQueryStudio.dcp dclTMSQueryStudio.dcp
Delphi XE7	TMSQueryStudio.dpk dclTMSQueryStudio.dpk	TMSQueryStudio210.bpl dclTMSQueryStudio210.bpl	TMSQueryStudio.dcp dclTMSQueryStudio.dcp