TurboDB 6

The Programmer's Guide to Powerful Database Applications

dataweb.
Smart Database Technologies

TurboDB 6

The desktop database engine to live with

by Peter Pohmann, dataweb

TurboDB is a full-featured multi-user database engine and a set of native components for accessing TurboDB database tables. TurboDB is available for Windows and .NET and supports Delphi and C++ Builder as well as Visual Studio.NET and all its programming languages.

TurboDB Components

Printed: November 2022 in Aicha, Germany

Table of Contents

1 TurboDB for VCL

1.1 Introduction

1.1.1 TurboDB Overview

TurboDB is a full-featured multi-user database engine and a set of data access components for accessing TurboDB database tables in your Delphi/C++ Builder application.

TurboDB for VCL

TurboDB Engine and TurboDB Components are 100% Delphi code. TurboDB Components are very close in use to the BDE data access components included in Delphi Professional and Delphi Enterprise.

Compared to these BDE components TurboDB offers the following advantages:

- Smaller executables, no additional dlls to deploy besides your exe-file
- No special installation and/or configuration needed
- Unicode is fully supported in strings and memos
- Tables can be encrypted
- Full-text indexing for very fast keyword search

Compared to the database client components for InterBase and MySQL, TurboDB offers a lot of advantages. TurboDB

- is much easier to install and configure
- offers table creation and altering within the IDE
- includes a table component to access database data without SQL statements
- is highly compatible to BDE, so you can migrate very quickly
- offers much more functionality via methods and properties

Requirements

You need one of the following Embarcadero development tools to work with TurboDB 6: Delphi 6/7/2005/2006/2007/2009/2010/XE Professional and above, C++ Builder 6/2006/2007/2009/2010/XE Professional and above.

TurboDB (like any other database access technology) will not work with any Personal or Open Edition of Delphi/C++ Builder because of missing base technology by Embarcadero.

Editions

TurboDB is available in different versions and editions. If you have further questions please read the FAQ or consult the TurboDB team at dataweb. TurboDB is based on the TurboDB database engine that is included in the TurboDB package.

TurboDB for .NET

dataweb offers also a database engine for .NET which is specifically written in C# and does not require any additional access rights to execute. See our homepage for more information on this product.

1.1.2 New Features

TurboDB 6 brings a lot of new features both in the core database engine and in the VCL component set. The new engine features including enhancements to TurboSQL can be found in the TurboDB Engine documentation. The changes to the component library are listed here:

- Support for all languages through collations on table and column level.
- Support for database back-up during regular database operation.

- Considerably enhanced full-text indexing and searching. In *TTdbTable.AddFulltextIndex2* you can now indicate a list of word separators. When searching via SQL the columns to search in can be specified.
- Choose to work with a incremental filter instead of a static filter where appropriate. Incremental filters are much faster when searching a large subset within a huge dataset. The advantages of static filters are still available by default.
- Support for drill-down functionality and manually modified filters through a new selection API in *TTdbDataSet*.
- *TTdbTable* has now an *IndexFieldNames* property like *TTable* that sorts after an arbitrary sequence of column names. I.e. there need not exist an index for this sorting order.
- The size of the cache used for table data can be specified.
- The VCL enumeration type *ftWideMemo* is supported.
- TurboDB Viewer' user interface has been reworked, the menu structure is much clearer now.
- TurboDB Viewer supports the backup feature and also a new function that creates a copy of a table in another database.
- The SQL editor in TurboDB Viewer has syntax highlighting and code completion.
- In TurboDB Viewer, a click on a column header sorts the table content after that column in ascending or descending order.
- And many more features on the database level.

See also

Upgrading

1.1.3 Upgrading a Major Version

There are some modifications to your existing project that you may have to do when upgrading from version 5 and below. When you compile your program the first time with the VCL components for TurboDB 5, you must open all forms in the IDE and click on the ignore button, whenever a message comes up related to TurboDB components. This is due to removed/replaced properties in the component set, but if you follow the instructions in this topic, your application will run as before.

From TurboDB 5

TTdbDataSet.Filter and TTdbDataSet.Locate

Because TurboDB 6 supports true collations and therefore the case sensitivity is now defined in the table column, the *FilterOption* value *foCaseInsensitive* and the *LocateOption* value *loCaseInsensitive* are without function. If you used these options, you must upgrade your database to level 6 and define the respective collation for the table or column.

TTdbTable.LangDriver

Because TurboDB 6 now supports true collations, you cannot use language drivers anymore. The property still exists for formal compatibility but is no more used. *TTdbTable* now has a new property *Collation*, which defines the default collation for its textual columns. If you have been using a language driver, upgrade the tables to level 6 and choose the corresponding collation for them.

TField for Links, Relations and Enumerations

Since links, relations and enumerations are now handled as Unicode strings in Delphi 2009 and above, corresponding fields must be of type *TWideStringField* instead of *TStringField*. When converting older programs you must delete fields of these types and re-create them.

From TurboDB 4

Directory Property

This property was already marked as obsolete in TurboDB 4 and has now been removed. Use the property Location instead.

Property FilterType

This property has been removed because it is no more necessary. With TurboDB 5 you can search for a condition and for keywords at the same time. When you first open a form with *TTdbTable* components on it in the Delphi/C++ Builder IDE, click Ignore in the dialog box telling you about the missing property.

Table Protection

Tables have now the *EncryptionMethod* property and the *Password* property instead of the Password property and the *Key* property. If you want to protect your table, you must now set the *EncryptionMethod*. In order to be compatible with TurboDB 4, this property must be set to *temClassic* if you have used a key or to *temProtection* if you did not use a key, just a password. If you have been using a key with TurboDB 4, please refer to the topic on the Password property to learn how to accommodate it. The password property is now a *WideString* instead of an *AnsiString* in TurboDB 4.

Related to this is a modification in the *OnPassword* event. Since keys are no more used in TurboDB 5, this parameter has been removed from the event's signature.

Full-text Indexes

You can continue to use the full-text searching code with your VCL components. But if you want to upgrade to the new table level 4 or if you want to profit from the new full-text index features like improved performance, ranking and maintained indexes, you must change your program code slightly but you will basically make it simpler.

With the new full-text search, there is no need to link to the keyword table anymore. The word filter still works the same way as before but instead of calling *AddFulltextIndex* you now call *AddFulltextIndex2*. And there is an additional *UpdateFulltextIndex* method, which needs only the full-text index name as its input.

Upgrading Issues on the Database Level

Further considerations especially for SQL statements.

1.1.4 Upgrading a Minor Version

After upgrading from a minor version of TurboDB you should process the following steps to ensure that your applications will be build with the new version.

1. Activate the components:
   If not already done during the setup of TurboDB, activate the components by running the activator tool in the installation directory. Ensure that the Delphi/C++Builder IDE is closed during activation. You can check the activation by right clicking any TurboDB component and take a look at the license information in the *About TurboDB* dialog.
2. Check the search path:
   Delphi/C++ Options
   The library path in Delphi/C++ should contain the full path to the installed TurboDB components or a environment variable $(TurboDB6). If $(TurboDB6) is included then check the Delphi environment variables whether *$(TurboDB6)* holds the full path to the installed TurboDB components. Remove all references to older versions of TurboDB
   Project Options
   Check your projects whether the search path differs from the IDE defaults. Add the full path to the installed TurboDB components, if not already set in the default options. Remove all references to older versions of TurboDB
3. Check version changes:
   Open the *readme* file to see the changes in this version. Process the modifications to your projects or data that are described there (if any).

4. Rebuild your project:
   To ensure that the upgraded components are used, you have to rebuild your projects, **compiling is not sufficient**.

1.1.5 First Steps

After the installation procedure as described in *readme.txt* the component palette of your Delphi or C++ Builder IDE contains an additional page called *TurboDB* and the contents of your online-help includes the *TurboDB* chapter.

A good point to start is the demo application included with your TurboDB package. It is to be found in the *samples* subdirectory of your installation folder. Just open the Delphi project file *tdbdemo.dpr* within the IDE and run it. Perhaps you have to adjust the database names of the TurboDB tables using the object inspector.

When you decide to start with your own project, you will need to create appropriate database tables. The TurboDB package includes visual tool called *TurboDB Viewer* for this and related tasks. It is located in the windows*bin* subdirectory of your installation folder. Another way is to use the table component editor within the IDE.

To start a new project using TurboDB you must:

1. Create a new project in the IDE.
2. Add a *TTdbTable* component to your form for each database table you need.
3. Open the component menu of each *TTdbTable* component and select *Properties* to define the table schema.
4. Set the *Active* property of the *TTdbTable* component to true, to open the database table.
5. Be sure to have the *windows\delphiX*, or *windows\cbX* subdirectory of your TurboDB installation folder included in the Delphi search path. (Replace the X by your version of Delphi, C++ Builder.)

Use the TurboDB as you are used to from the BDE database components. This help documents each method and each property available with TurboDB.

1.1.6 Installing the VCL Edition

1.1.6.1 Windows Installation

The # character is used a placeholder for the internal version number of Rad Studio, Delphi or C++ Builder. See list below to find the right number for your compiler.

1. Shut down any running instances of Rad Studio, Delphi or C++ Builder.
2. Start TurboDBVCL6.msi. The setup program will uninstall older versions of TurboDB, installs the component package and registers the TurboDB palette within your IDE. If the automatic uninstallation of the old version fails, please manually remove it from the windows control panel.
3. Start your compiler. You will find a new component palette page called TurboDB that holds the new components. If you do not see this component palette then open *Component/Install packages*. Check the entry "dataweb TurboDB 6 VCL" or use the "Add..." button to install the TurboDB 6 design time package "dclturbodb6d##.bpl".
4. Select *Tools* > *Environment* > *Library* and add the TurboDB components path suitable for your compiler to the library path, if it isn't there already.
5. The setup program merges the TurboDB help directly into the help system of your compiler.

If this wasn't successful you can access the help file "TurboDB6.chm" in the "Documentation" directory of the TurboDB Installation.

6. Alternatively you can read the Online Documentation on our homepage.
7. If you experience any problems with the installation, please report to the dataweb support team.

Compiler Versions and Pathes

Internal Version No	Compiler	Path to TurboDB 6 Components	Platform
6	Delphi / C++ Builder 6	<TurboDB 6 InstallDir>\Lib\Delphi6	Win32
7	Delphi 7	<TurboDB 6 InstallDir>\Lib\Delphi7	Win32
9	Delphi 2005	<TurboDB 6 InstallDir>\Lib\Delphi9	Win32
10	Delphi 2006	<TurboDB 6 InstallDir>\Lib\Delphi10	Win32
11	Delphi 2007	<TurboDB 6 InstallDir>\Lib\Delphi11	Win32
12	RAD Studio / Delphi / C++ Builder 2009	<TurboDB 6 InstallDir>\Lib\Delphi12	Win32
14	RAD Studio / Delphi / C++ Builder 2010	<TurboDB 6 InstallDir>\Lib\Delphi14	Win32
15	RAD Studio / Delphi / C++ Builder XE	<TurboDB 6 InstallDir>\Lib\Delphi15	Win32
16	RAD Studio / Delphi / C++ Builder XE 2	<TurboDB 6 InstallDir>\Lib\Delphi16\win32 <TurboDB 6 InstallDir>\Lib\Delphi16\win64	Win32 Win64
17	RAD Studio / Delphi / C++ Builder XE 3	<TurboDB 6 InstallDir>\Lib\Delphi17\win32 <TurboDB 6 InstallDir>\Lib\Delphi17\win64	Win32 Win64
18	RAD Studio / Delphi / C++ Builder XE 4	<TurboDB 6 InstallDir>\Lib\Delphi18\win32 <TurboDB 6 InstallDir>\Lib\Delphi18\win64	Win32 Win64
19	RAD Studio / Delphi / C++ Builder XE 5	<TurboDB 6 InstallDir>\Lib\Delphi19\win32	Win32 Win64

		<TurboDB 6 InstallDir>\Lib\Delphi19\win64	
20	RAD Studio / Delphi / C++ Builder XE 6	<TurboDB 6 InstallDir>\Lib\Delphi20\win32 <TurboDB 6 InstallDir>\Lib\Delphi20\win64	Win32 Win64
21	RAD Studio / Delphi / C++ Builder XE 7	<TurboDB 6 InstallDir>\Lib\Delphi21\win32 <TurboDB 6 InstallDir>\Lib\Delphi21\win64	Win32 Win64
22	RAD Studio / Delphi / C++ Builder XE 8	<TurboDB 6 InstallDir>\Lib\Delphi22\win32 <TurboDB 6 InstallDir>\Lib\Delphi22\win64	Win32 Win64
23	RAD Studio / Delphi / C++ Builder 10 Seattle	<TurboDB 6 InstallDir>\Lib\Delphi23\win32 <TurboDB 6 InstallDir>\Lib\Delphi23\win64	Win32 Win64
24	RAD Studio / Delphi / C++ Builder 10.1 Berlin	<TurboDB 6 InstallDir>\Lib\Delphi24\win32 <TurboDB 6 InstallDir>\Lib\Delphi24\win64	Win32 Win64
25	RAD Studio / Delphi / C++ Builder 10.2 Tokyo	<TurboDB 6 InstallDir>\Lib\Delphi25\win32 <TurboDB 6 InstallDir>\Lib\Delphi25\win64	Win32 Win64
26	RAD Studio / Delphi / C++ Builder 10.3 Rio	<TurboDB 6 InstallDir>\Lib\Delphi26\win32 <TurboDB 6 InstallDir>\Lib\Delphi26\win64	Win32 Win64

1.1.6.2 Licensing and Activation

If you are using the VCL component libarary you need to include the file *TdbLicense* in each project to activate the features you have licensed. Without this unit, your program will throw an exception, when you first try to open a database connection. Including TdbLicense is very easy:

- In Delphi just add the unit *TdbLicense* to the uses clause of any unit in your program.
- In C++ Builder insert the line *#pragma link TdbLicense* in the main C++ file of your application. Additionally check the project option "Build with runtime packages" (default in C++ Builder). Either uncheck this option or add the lib file "turbodb6dX.lib" (turbodb6dX.a for x64) to your project.

As long as you didn't receive a license file and you didn't activate your TurboDB installation, your applications will run for 30 days counted from the day of the TurboDB 6 installation. After that period, your application will throw an exception when you try to open a database.

Activation is the process of converting a trial edition of TurboDB into a licensed edition, which you can use to build commercial programs. In order to activate your trial edition of TurboDB you need an license file which you receive upon purchasing a license.

Let us assume you have completed your application using the trial edition of TurboDB and then you have purchased a license. You will receive an e-mail containing the license file.

1. Run Activator which resides in the bin subdirectory of your installation. Under **Windows Vista or later** you have to run the activation utility by using "Run as Administrator". To do this you just right-click activator.exe and select the option from the drop down menu.
2. Check if the TurboDB directory is correct, enter the path to the license file and press Ok. The activator will modify some files of your TurboDB installation.
3. You are done. Again, please don't forget to use the unit *TdbLicense* in any TurboDB application you create. It is a very small unit and contains the encrypted licensing information, that is necessary to create a commercially used program.

1.1.7 Demo Programs

TurboDB for VCL comes with a set of demo programs. Those demos are installed in the *All Users/ Public* respectively in the folder *dataweb/TurboDB VCL 6/Demos*.

Company: Master-detail data sets

ToDoList: Queries, enumerations

Fulltext: Full-text indexing and searching

Relationship: Link and relation columns

Drill-Down: Selection API

Images: Blobs and the *TTdbBlobProvider* component

1.1.7.1 Company Sample

Platform: Windows

Language: Object Pascal

Compiler version: 6 and above

Description: Manages employees and their departments in two linked data grids. Offers filtering, search, sorting and export functionality. *Company* is a VCL application, so you can compile it with any compiler that supports TurboDB.

Demonstrates: Master detail relationship, link fields, use of the *TTdbBatchMove* component

1.1.7.2 ToDoList Sample

Platform: Windows

Language: C++

Compiler version: 6

Description: Small & easy action item management with short and long description, deadline, importance and state.

Demonstrates: Queries, enumeration data type in database tables, memos, editing ordered result sets, use of look-up fields, use of *TTdbEnumValueSet.*

1.1.7.3 Fulltext Sample

Platform: Windows

Language: Object Pascal

Compiler version: 6 and above

Description: Allows indexing text files in arbitrary directories. Then you can search for keywords and combinations of keywords. *Fulltext* is a VCL application, so you can compile it with any compiler that supports TurboDB.

Demonstrates: Full-text indexes and full-text searching

1.1.7.4 Relationship Sample

Platform: Windows

Language: Object Pascal

Compiler version: 6 and above

Description: Manages departments, locations and employees. You can assign employees to departments and departments to locations. *Relationship* is a VCL application, so you can compile it with any compiler that supports TurboDB.

Demonstrates: Working with link and relation fields for creating one-to-many and many-to-many master-detail views.1 Chained master-detail views.

1.1.7.5 Drill Down

Platform: Windows

Language: Object Pascal

Compiler version: 6 and above

Description: Displays a table and lets the user select and unselect rows iteratively. Selected records are marked in the indicator column. The user can add and remove selections either through filter conditions or manually. Note that due to the limited capabilities of the VCL DBGrid component, the selection is not always correctly shown. Click the refresh button to update the selection indication, when the selection seems to be wrong.

Demonstrates: Working with selections either through filter conditions or through manual addition and removal.

1.1.7.6 Images

Platform: Windows

Language: Object Pascal

Compiler version: 6 and above

Description: Stores images in a database table, displays them and lets the user add, delete and modify them.

Demonstrates: Working with blobs in the database and with the blob provider component.

See also

TTdbBlobProvider

1.1.8 Support

1.1.8.1 Support

dataweb provides technical support for TurboDB in the Internet.

FAQ

Is your question in the frequently asked questions on the Web? Have a look at http://www.turbodb.de/en/support/faq_general.html.

Web Site

Visit the TurboDB Web site at *http://www.turbodb.de*.

Forum

dataweb hosts an English spoken discussion forum which is maintained by dataweb staff.

E-Mail

If you have questions or problems that are not answered on the Web Site or in the forum then send a mail to the dataweb support team under *support@dataweb.de*.

Professional Services

We offer you our expertise in consulting & programming in areas like TurboDB and general Windows programming in C++, Delphi, C#. We are specialized in storage and query implementation, design and implementation of scripting languages, evaluation of large amounts of time series data, automation and system engineering using editable diagrams: *http://www.dataweb.de/en/products/diagramming.html*

1.1.8.2 Versions and Editions

Version 6 of TurboDB is available for Delphi/C++ Builder on Windows.

Standard Edition

The Standard Edition allows up to 63 tables opened at the same time and shares tables between applications.

Professional Edition

Like the standard edition but additionally provides SQL support. For the Embarcadero tools this means a *TTdbQuery* component is included.

Evaluation Edition

Identical to the professional edition, but limited to 30 days. Applications built with this version as well as the design-time support will cease to work after this period.

1.2 VCL Component Library

1.2.1 Overview

This chapter describes the component library for Delphi and C++ Builder. For detailed information on the database engine, its features, or the TurboSQL dialect, please refer to the chapter TurboDB Engine.

1.2.2 Developing with TurboDB

1.2.2.1 Working with Tables

1.2.2.1.1 Creating a Table at Design-time

There are different ways to create a database table. You can use the features built-in in the Delphi IDE, you can use the visual TurboDB Viewer or you can use the text-based TurboDB Workbench utility.

To create a database table using the built-in IDE features you have to:

1. Drop a *TTdbTable* component onto a form or data module
2. Right-click on it and select the *New Table* command
3. Edit the table columns and build the table.

To create a database table using the visual TurboDB Viewer application:

1. Open the TurboDB Viewer from your Delphi tools menu or the start menu.
2. Select the *Database/New/Table...* command
3. Edit the table columns and build the table.

1.2.2.1.2 Creating a Table at Run-time

The *CreateTable* method of the *TTdbTable* components creates a new database table at runtime. The table structure is determined by the *FieldDefsTdb* property of the table. If you want to create a completely new table, you must clear *FieldDefsTdb* first and then add the *TdbFieldDefs* you want to assign to the new table.

TurboDB also supports the standard mechanism based on the *FieldDefs* property of *TDataSet*. While this is a great way to assert compatibility, *FieldDefsTdb* offers a greater control and makes available special features and field types not supported in the standard.

To clear the FieldDefsTdb,

- Call *FieldDefsTdb.Clear*.

To add a *TdbFieldDef* to the *FieldDefsTdb* of the table,

1. Call *FieldDefsTdb.Add*,
2. Set the properties of the *TdbFieldDef* returned by this function.

To create a database table at runtime,

1. Set the *FieldDefsTdb* property according to the fields the new table should have,
2. Set the *TableLevel* property,
3. Set the *Password* property, if you want your table to be protected,
4. Set the *DatabaseName* and the *TableName* property
5. Call the *CreateTable* method.

Note: If there is already a table with this name, an exception will be raised.

The following code creates a new table with three columns:

```
TdbTable2.Close;
TdbTable2.FieldDefsTdb.Clear;
TdbTable2.FieldDefsTdb.Add('Word', dtString);
TdbTable2.FieldDefsTdb.Add('Count', dtSmallInt);
with TdbTable2.FieldDefsTdb.Add('RecordId', dtAutoInc) do Specification
:= 'Word';
TdbTable2.TableName := 'index';
TdbTable2.TableLevel := 6;
TdbTable2.CreateTable;
TdbTable2.Open;
```

To specify calculated table columns

1. When defining the columns of the table through the *TTdbFieldDef* objects, assign an expression to the *CalcExpression* property.
2. If the expression is to be used for calculating a new value for the column each time the row data changes, set the *InternalCalcField* property to *True*. If the expression shall be used to calculate a default value for the column, set *InternalCalcField* to *False*.

1.2.2.1.3 Altering a Table at Run-time

Use the *AlterTable* method to restructure a database table at runtime. *AlterTable* uses the *FieldDefsTdb* property to determine the field definitions of the altered table. TurboDB keeps all field values if possible, even if the field type is changed or the field is renamed. The structure of the altered table is set in the same way as for creating the table. Altering a table can also be used to change the level of a table or to modify its password or key.

Note: Altering a table may result in loosing data. Especially if you delete columns of a table or if you shorten alphanumeric fields, the data stored in the table is lost.

Note: If there should be any problem during the restructuring and your program crashes, your data is not lost. The original tables are renamed to ~TableName before the restructuring begins. If a problem occurs you just have to rename this file to restore all your original data.

The example shows how to change the encryption key of a table at runtime:

```
TdbTable4.EncryptionMethod := temBlowfish;
TdbTable4.Password := 'dataweb';
TdbTable4.AlterTable;
```

1.2.2.1.4 Selections and Drill-Down

TurboDB provides an extension over the regular filters the VCL offers. A selection is a subset of records, which can be manipulated by adding and removing single records, applying filter conditions and word filter conditions. The subset can be used for example to manage a multi-selection of records in a grid, to implement drill-down capability or to realize the functionality of the SQL operators UNION, INTERSECT and EXCEPT on the *TTdbTable* level. Selections can also be used as filters.

To filter out the records with record no 54821 and 897003:

```
MyTable.AddToSelection(54821);
MyTable.AddToSelection(897003);
MyTable.Filtered := True;
```

To filter out all cars that are from BMW and have red color incrementally (drill-down):

```
MyTable.Filter = 'Manufacturer = ''BMW''';
MyTable.Filtered := True;
// Now all BMWs are shown
MyTable.AddToSelection('Color = ''red''');
// Now all red BMWs are in the selection
MyTable.ActivateSelection;
// Now all red BMWs are shown
MyTable.Filtered := False;
// All cars are shown again.
```

To delete all red BMWs incrementally:

```
MyTable.ClearSelection;
MyTable.AddToSelection('Manufacturer = ''BMW''');
MyTable.IntersectSelection('Color = ''red''');
MyTable.Last;
while not MyTable.BOF do begin
  if MyTable.IsSelected(MyTable.RecNo) then
    MyTable.Delete;
  MyTable.Prior;
end;
```

When a data set has a filter set but the Filtered property is false, the records that satisfy the filter condition make up the current selection. It can be modified using the *AddToSelection*, *RemoveFromSelection* and *IntersectSelection* methods. The *FindFirst* and *FindNext* methods can be used to browse through the current selection.

See also

DrillDown Demo Program
AddToSelection method
RemoveFromSelection method
IntersectSelection method
ActivateSelection method
IsSelected method
ClearSelection method

1.2.2.2 Using Indexes

1.2.2.2.1 Creating an Index at Design-time

There are some ways to create an index. You can use the features built-in in the Delphi IDE, you can use the visual TurboDB Viewer or you can use the text-based TurboDB Workbench utility.

To create an index using the build-in IDE features you have to:

1. Drop a TTdbTable component onto a form or data module
2. Right-click on it and select the *Properties* command
3. Define the index and build it.

To create an index using the visual TurboDB Viewer application:

1. Open the TurboDB Viewer from your Delphi tools menu or the start menu.
2. Open a table using the *Database/Open/Table* command
3. Select the *Table/Properties...* command
4. Define the index and build it.

1.2.2.2.2 Creating a Full-text Index at Design-time

A full-text index is a data structure that enables you to find a record based on keywords in any column of the table very fast. TurboDB realizes this features by creating a helper table called dictionary, which contains all the keywords. In older tables (level 3 and below) the relationship between the dictionary and the data table is established via a relation field. In current tables, there is a special full-text index, which implements this relationship in a much faster an maintainable way.

To create a new type full-text index with TurboDB Viewer (table level 4 and above):

1. Open TurboDB Viewer and connect to the database you want to create the full-text index in.
2. Create the dictionary table if you do not yet have one. This table must have a string column large enough to hold your keywords (e.g. 40 characters), a byte column, which will later store the relevance of this keyword and an AutoInc field with indication set to the keyword field.
3. Select *Table/Properties...* to display the properties window for the table, the full-text index will apply to. Note that new type full-text indexes are only available for table level 4 and above.
4. Switch to the index page and click on the *New* button.
5. Define a name for the index and select Full-text as the index type.
6. On the Full-text page, select all the columns you want to insert into the full-text index and select the dictionary table you want to use (the one you created in the 2nd step).
7. Leave the minimum relevance as it is and click Ok. The full-text index will be created.

To create an old type full-text index with TurboDB Viewer (table level 3 and below):

1. Open TurboDB Viewer. Usually it can be found in your Tools menu. The application is located in the bin subdirectory of your TurboDB installation folder.
2. Open the table you want to create the full-text index for using the command *Database/Open/Table...*
3. Open the full-text index wizard with *Table/Create Full-Text Index...*
4. On the first page select a name for the new dictionary table that will accept the keywords. Then enter a name for the keyword column within this table. The third item is the maximum length of the keywords. Keyword longer than this value will be cut.
5. On the second page you can choose, which columns in the original table will be scanned. You may add all columns to the full-text index but very often you will add only a few fields.
6. Since searching for words like and, in or a in a full-text index does not make much sense, you should try to eliminate those keywords form the full-text index. You can do this in two ways. First, you may decide to throw away keywords, which occur more often than a given limit, e.g. 100. Second, you can provide a text file that contains the words you don't want to be added to the full-text index. Both techniques will help to keep the full-text index small, will speed up searching and will avoid useless hits. The appropriate settings are done on the third page of the wizard.
7. On the forth page of the wizard you are ready to create the full-text index. Since this action involves scanning all fields of all records in the table, it might take quite a time until completion.

1.2.2.2.3 Creating a Full-Text Index at Run-Time

Only new type full-text indexes can be created at run-time, this means the table must have level 4 or higher.

To create a new table with full-text index with the TTdbTable component:

1. Create a *TTdbTable* component for the table, you want to create.
2. Define all the properties (e.g. FieldDefs) necessary for the new database table.

3. Add one *TTdbFullTextIndexDef* for each full-text index you need to the *FullTextIndexDefs* property of the table.
4. Call the *CreateTable* method of the *TTdbTable* component.

To add a full-text index at run-time with the *TTdbTable* component:

1. Create a *TTdbTable* component for the table, you want to add the full-text index to.
2. Call the *AddFullTextIndex* method, passing the parameters as described in the previous section.

To add a full-text index at run-time using SQL:

1. Create a *TTdbQuery* component and set the SQL text to:
```
CREATE FULLTEXTINDEX ON <Table-Name> DICTIONARY <Dictonary-Table-Name> (<Field1>, <Field2>, <Field3>, ...)
```
2. Call the *ExecSQL* method of the *TTdbQuery* component.

See also

CREATE FULLTEXTINDEX

1.2.2.2.4 Using a Full-text Index at Run-Time

If you have created a full-text index you can use it to locate records containing one or more given keywords in just a few milliseconds.

To use a full-text index at run-time, you have to:

1. Place a *TTdbTable* component for the table your are searching in on your form.
2. At design-time or at run-time set the *WordFilter* property to your desired full-text expression, e.g. `WordFilter := 'Embarcadero';`
   The syntax for keyword filter expressions is explained in "Full-Text Search-Conditions".
3. Now you can work with this filter like you are used to from normal filters. You may employ the *FindFirst*, *FindNext*, *FindPrior* and *FindLast* methods to locate the record or you can just set the *Filtered* property to True.
4. You may also combine word filters with standard filter-conditions by setting the *WordFilter* property and the *Filter* (or *FilterW*) property at the same time.

Annotation

Only old type full-text indexes (table level 3 and lower) need an additional table component for the dictionary table. Set the *FullTextTable* property to the component for the dictionary table.

1.2.2.2.5 Updating or Reparing an Index

Sometimes, mainly because of crashes or program termination during debugging, an index might go out of sync with the table. You will notice this, when the number of records in the table is less then expected, when this index is set or by doing a check of the table in TurboDB Viewer. If the index is no more correct, you can repair it by just re-building it:

Rebuild an index with TurboDB Viewer:

1. Connect to the database and select the table you want to rebuild the index for.
2. Select *Table/Maintain...* in the main menu.
3. If you want to check the table first, click on the *Start* button and wait until the analysis is finished. It will display problems found with the table. If there is something about an index, the index must be re-built.
4. Check the *Rebuild all indexes* check box in the Repair Options group and click *Apply*. TurboDB Viewer will now rebuild all indexes of the table. Which might take a few minutes if the table is large.

Rebuild an index at run-time with a TTdbTable component:

1. Create a *TTdbTable* component for the table you want to rebuild the index for.
2. If the index is a normal index call the *UpdateIndex* method, if it is a full-text index, call the *UpdateFulltextIndex* method.

Rebuild an index at run-time using SQL:

1. Create a *TTdbQuery* component for the database the index is in.
2. Set the SQL text property to
```
UPDATE INDEX <Table-Name>.<Index-Name>
or in case of a full-text indes to
UPDATE FULLTEXTINDEX <Table-Name>.<Full-text-Index-Name>
```
3. Call the ExecSQL method on the *TTdbQuery* component.

1.2.2.3 Importing and Exporting Records

1.2.2.3.1 Executing a Batch Move

The BatchMove component is a powerful and flexible way to transfer records from and to different data sources. You can use different flat files and all *TDataSet* descendants as data sources.

To execute a batch move:

1) Set the TdbDataSet property to the TTdbTable component where you want to import from or to export to.

2) Set the FileName or the DataSet property to the file or the data set of the other data source.

3) If the other data source is a file, set the FileType property according to the file format of the other data source.

4) If the other data source is a text file (i.e. FileType has been set to tffSDF), set Quote, Separator and ColumnNames to the appropriate values.

5) If the other data source is a text file or a dBase file, set CharSet to the appropriate value.

6) Set the Direction property to bmdImport, if you want to transfer records from the other data source into the TurboDB table. Set it to bmdExport, if you want to create a file with records of your TurboDB table.

7) Set the Mode property to the kind of batch move you want to perform.

8) Add the column mappings between the source and target to the Mappings property.

9) Set the Filter property to restrict the import or export to a subset of the source data set.

10) Register an event handler for the OnProgress event to show the progress of the batch move process and to offer the user a way of interrupting the operation.

11) Call the Execute method to start the batch move.

1.2.2.4 Migrating from BDE

1.2.2.4.1 Porting a BDE application to TurboDB

Porting BDE applications that use Paradox or dBase files to TurboDB is easy because TurboDB is based on components very similar to the BDE components. *TTdbDataSet* replaces *TBDEDataSet*, *TTdbTable* replaces *TTable*, *TTdbQuery* replaces *TQuery*, *TTdbDatabase* replaces *TDatabase* and *TTdbBatchMove* replaces *TBatchMove*. These TurboDB components offer the same properties, methods and events as the BDE components, so you can re-use most of your existing source code with TurboDB.

Basically, porting your BDE application works like this:

a) Using the TurboDB Table convert assistant (only available in Delphi/C++ versions with integrated BDE)

1. Install TurboDB within your Delphi IDE.
2. Make a back-up of your BDE project and open it in the Delphi IDE.
3. Place a *TTdbTable* object near each *TTable* component in your application.
4. Right click on each *TTdbTable* component and select the "Convert BDE Table" command.
5. Select the *TTable* component you want to convert.
6. TurboDB will create a TurboDB table and copy the contents of the BDE table. It will also copy the properties of the BDE table component.
7. Rename all *TTable* components and give the TTdbTable components the original name of the TTable components.
8. If your project contains *TQuery* objects, replace them by *TTdbQuery*. You can just copy the SQL property value if necessary. See "TurboSQL vs. Local SQL" for the differences between the SQL dialects.
9. If your project contains a TDatabase and/or TSession object, replace it by a TTdbDatabase object.
10. If your project contains a TBatchMove component, use a TTdbBatchMove component instead.
11. Now compile and run your application. If you get error messages from the compiler or from your application, please consult "Differences between BDE and TurboDB".

b) Manual conversion

1. Open the TurboDB Viewer and convert all BDE tables needed in your project to TurboDB (Tools/Import Tables). Check within the Viewer if all tables were converted properly.
2. Install TurboDB within your Delphi IDE.
3. Make a back-up of your BDE project and open it in the Delphi IDE.
4. Place a *TTdbTable* object near each *TTable* component in your application and open the corresponding TurboDB table as converted in step 1.
5. At the *TTdbTable* set the following properties: AutoCalcFields, Exclusive, Filter, FilterOptions, ReadOnly, IndexName, Filtered.
   Look around the the correspondent properties at the *TTable.*
6. Proceed likewise with the events of the *TTdbTable* component.
7. Rename all *TTable* components and give the *TTdbTable* components the original name of the *TTable* components.
8. If your project contains *TQuery* objects, replace them by *TTdbQuery*. You can just copy the SQL property value if necessary. See "TurboSQL vs. Local SQL" for the differences between the SQL dialects.
9. If your project contains a *TDatabase* and/or *TSession* object, replace it by a *TTdbDatabase* object.
10. If your project contains a *TBatchMove* component, use a *TTdbBatchMove* component instead.
11. Now compile and run your application. If you get error messages from the compiler or from your application, please consult "Differences between BDE and TurboDB".

1.2.2.4.2 Differences between BDE and TurboDB

- TurboDB uses its own file format. This means you must convert your tables as described in "Porting BDE Applications to TurboDB".
- TurboDB does not support SQL servers. If you want to port an application to TurboDB that uses BDE to access a SQL server you have to create TurboDB table files for your data as described in "Porting BDE Applications to TurboDB".
- TurboDB does not support some of the field types available with the BDE. These field types are: ftWord, ftCurrency, ftBCD, ftParadoxOle, ftDBaseOle, ftTypedBinary, ftCursor, ftFixedChar, ftADT, ftArray, ftReference, ftDataSet, ftOraBlob, ftOraClob, ftVariant, ftInterface, ftIDispatch
- There is no *TSession* object with TurboDB. Like dbExpress and other Delphi database technologies TurboDB uses connection objects (*TTdbDatabase*) for managing the connection.
- The localization support of TurboDB works by setting a windows collation.

1.2.2.4.3 Beyond the BDE

TurboDB offers features to your applications that exceed what you can do with the BDE:

- Password-protect or even encrypt your tables
- Find records by keywords in any field within milliseconds using and, or and not operators
- Much more flexible batch move component to transfer from and to any data set and to MyBase data packets
- Add, modify and remove table columns at run-time using the powerful AlterTable method
- Use hyphens and German Umlauts in table and column names
- Have table columns with an enumeration data type
- Very user-friendly visual table viewer and table designer with powerful import/export wizard
- Special column types for implementing convenient one-to-many and many-to-many relationship between tables

1.2.2.5 Localizing your application

1.2.2.5.1 Translating the User Interface

TurboDB produces three kinds of error messages: Context messages, error descriptions and error reason messages. All message texts can be found in the Object Pascal unit TdbMessages.pas. This unit is delivered as source code with every edition of TurboDB. It currently contains all message texts in English and German. The default language is English.

If you want to use the German messages, you must:

1) Define the symbol GERMAN in the conditional defines in the project options.

2) Be sure to include TdbMessages.pas in the unit search path.

3) If you are using C++ Builder, you must include the unit TdbMessages.pas in your project, too.

If you want to use the messages in another language, you can translate them by yourself and include them in the unit TdbMessages.pas, guarded by the appropriate conditional define. When you send us your translation of the messages, we will include them in the sources. This way others can profit from your work and you need not adjust the source code every time, you get a new version.

1.2.2.5.2 Localizing String Comparison

Countries that use other character sets unlike the English or the German need customized collate sequences.

Language drivers as used in TurboDB 5 and before are obsolete, do not use them anymore. TurboDB 6 now supports Windows collate sequences which can be set on table and/or field basis. See collations and TTdbTable.Collation.

1.2.2.6 Miscellaneous

1.2.2.6.1 Storing ANSI and UnicodeString

In Delphi and C++ Builder up to version 2007 the standard string is *AnsiString* which can easily be stored in string columns and memo columns. In these versions Unicode strings must be declared as WideString and some thoughts are required on how to store them in the database correctly. With Delphi and C++ Builder starting from version 2009, the standard string is a Unicode string and difficulties come up when you store them as (ANSI) strings or memos.

When Storing Unicode Strings with Delphi/C++ Builder 2007 and Below

The TurboDB Engine supports Unicode in *WideString* columns and in *WideMemos*. Working with those data types in a VCL application however contains some pitfalls. The VCL almost everywhere uses *AnsiStrings* in its functions and components and converts Unicode strings automatically and without warning to this type. As a consequence the database controls also only receive and pass *AnsiStrings*. This means there is no way to enter and/or view Unicode data from your database in VCL data controls.

This is what you can do with the Unicode data in your TurboDB database within a VCL application:

- Show and edit strings in normal controls, e.g. *TEdit*. Those controls work with Unicode correctly but you will have to write your own updating and posting logic, since normal controls have no *DataSource* link.
- Work with the Unicode data internally. You can read and write *WideStrings* from and to a data set component.
- When accessing Unicode data from a *TdbDataSet* be careful not to use the *TField.AsString* property but the Value or the *AsWideString* property.
- Query parameters (i.e. items of the *TTdbQuery.Params* collection) can only be set via the *TParam.Value* property.
- If you are working with Unicode memos you will notice quickly that VCL does not offer a data type for this kind of columns. Unicode memos are represented by the *ftBlob* field type and the custom field class *TTdbBlobField*, which offers a *AsWideString* property.
- The *Filter* property in VCL components as well as the SQL property are both of type *AnsiString*. Therefore you cannot use them for searching for Unicode strings. The TurboDB database components therefore have additional *FilterW* and *SQLW* properties which are of type *WideString*.

When Storing ANSI Strings with Delphi/C++ Builder 2009 and Above

Many applications that need not be localized for other languages can still easily store its text data within string and memo columns, which have two advantages:

- Better compatibility with older applications
- Less storage requirements and therefore also better performance.

However storing (Unicode) strings in (ANSI) string and memo columns requires a conversion of the Delphi variable to an *AnsiString*. You must be sure that the content of the *UnicodeString* variable does not contain true Unicode characters or you will loose some of the information in the

string when storing it to the database.

1.2.2.6.2 Protected Database Tables

When opening protected database tables you need to provide a password. There are different ways to do this.

a) Set the *EncryptionMethod* and *Password* property of *TTdbTable* before you call the Open method.

b) Specify an even handler for *TTdbDatabase.OnPassword*. This event handler will be called, when TurboDB is not able to open a database table due to protection.

c) Include the unit *TdbPasswordDlg* into your project. This unit contains a simple password dialog, which will register with the database access components and show up each time a password is required. This password dialog is the one, which is also used at design-time.

See also

Data Security

1.2.2.6.3 Read-Only Tables and Databases

You can set a database and/or a table to read-only (*ReadOnly* property) to prevent any modifying access to it. Even if a database table is opened read-only, TurboDB has to create a net-file for it, because otherwise a second application could open the database in read-write mode and cause an access conflict.

If the database is stored in a location where no write access is possible (write-protected directory, DVD etc.) you must open the database exclusively and read-only. In this case TurboDB will not try to create *net* and *mov* files.

In the rare case, where two applications need concurrent access to the database in a read-only location, you can call *SetSharedReadOnly(True)* instead of setting the *Exclusive* property. In this mode, TurboDB does not create *net* and *mov* files but opens the database in a shared mode. By calling this method, **the application must guarantee that no other application will modify the database** as TurboDB by itself cannot detect modifications in this mode.

1.2.3 VCL Components Reference

1.2.3.1 VCL Components

The following components are located on the TurboDB tab within your Delphi IDE:

TTdbTable

TTdbBatchMove

TTdbDatabase

TTdbQuery (professional edition only)

TTdbEnumValueSet

TTdbBlobProvider

The base class of TTdbTable and TTdbQuery is TTdbDataSet, which in turn is a TDataSet descendant. Therefore all properties, methods and events of TDataSet apply to the corresponding TurboDB components.

Other TurboDB classes:

ETurboDBError

TTdbFieldDef

TTdbFieldDefs

TTdbForeignKeyDef

TTdbForeignKeyDefs

TTdbFulltextIndexDef

TTdbFulltextIndexDefs

This document assumes that you are familiar with the Delphi development environment and know how to use the standard data access, and data control components.

1.2.3.2 ETurboDBError

Describes a database error specific to TurboDB.

Unit

TdbDataSet

Description

When TurboDB detects an error which is not identical to the corresponding BDE error, it throws a *ETurboDBError* exception in place of *EDatabaseError.* This exception provides a more detailed error description including a reason for the error.

1.2.3.3 ETurboDBError Hierarchy

Hierarchy

TObject

|

Exception

|

EDatabaseError

|

ETurboDBError

1.2.3.4 ETurboDBError.Properties

In ETurboDBError

TdbError

Reason

Derived from EDatabaseError

(check Embarcadero documentation for more information)

HelpContext

Message

1.2.3.5 ETurboDBError.Reason

Indicates the reason for an error.

Delphi syntax:

```
property Reason: SmallInt;
```

C++ syntax:

```
__property short Reason = {read=FReason, write=FReason, nodefault};
```

Description

Examine Reason to determine the reason code for a TurboDB Engine error.

1.2.3.6 ETurboDBError.TdbError

Indicates the TurboDB Engine error code.

Delphi syntax:

```
property TdbError: SmallInt;
```

C++ syntax:

```
__property short TdbError = {read=FTdbError, write=FTdbError,
nodefault};
```

Description

Read TdbError to determine the TurboDB Engine error code. The meaning of this code is described in Error Description Codes.

1.2.3.7 TTdbDataSet

Encapsulates TurboDB functionality for descendant data set objects.

Unit

TdbDataSet

Description

TTdbDataSet is a data set object that defines TurboDB functionality for a dataset. Applications never use *TTdbDataSet* objects directly. Instead they use the descendants of *TTdbDataSet*, such as *TTdbTable or TTdbQuery*, which inherit its database-related properties and methods.

Developers who create custom dataset components that work through TurboDB may want to derive them directly from *TTdbDataSet* to inherit all the functionality of *TTdbDataSet* and the TurboDB-related properties and methods of *TTdbDataSet*.

1.2.3.8 TTdbDataSet Hierarchy

Hierarchy

TObject

|

TPersistent

|

TComponent

|

TDataSet

|

TTdbDataSet

1.2.3.9 TTdbDataSet Methods

In TTdbDataSet

ActivateSelection

AddToSelection

GetEnumValue

IntersectSelection

IsSelected

Locate

Lookup

RemoveFromSelection

Replace

SaveToFile

Derived from TDataSet

(check Borland/CodeGear/Embarcadero documentation for more information)

1.2.3.10 TTdbDataSet Properties

In TTdbDataSet

DatabaseName

FieldDefsTdb

Filter

FilterMethod

Filtered

FilterOptions

FilterW

Version

Derived from TDataSet

(check Embarcadero documentation for more information)

1.2.3.11 TTdbDataSet Events

In TTdbDataSet

OnProgress

OnResolveLink

Derived from TDataSet

(check Embarcadero documentation for more information)

1.2.3.12 TTdbDataSet.ActivateSelection

Activates the current selection.

Delphi syntax:

```
procedure ActivateSelection;
```

C++ syntax:

```
void __fastcall ActivateSelection(void);
```

Description

Activating the current selection is like setting a filter: Only the records in the selection are then shown. *ActivateSelection* sets the *Filtered* property to True if it was not set yet.

See also

Selections and Drill-Down

1.2.3.13 TTdbDataSet.AddToSelection

Adds records to the selection.

Delphi syntax:

```
procedure AddToSelection(RecordNo: TTdbRecordNo); overload;
procedure AddToSelection(const Filter: UnicodeString); overload;
```

C++ syntax:

```
void __fastcall AddToSelection(TTdbRecordNo RecordNo);
void __fastcall AddToSelection(System::String Filter);
```

Description

Calls the first overload, if you want to add a specific record to the selection. Call the second one, if you want to add all records to the selection, that comply with a filter condition.

See also

Selections and Drill-Down

1.2.3.14 TTdbDataSet.ClearSelection

Removes all records from the current selection.

Delphi syntax:

```
procedure ClearSelection;
```

C++ syntax:

```
void __fastcall ClearSelection(void);
```

See also

Selections and Drill-Down

1.2.3.15 TTdbDataSet.CreateBlobStream

Returns a TBlobStream object for reading or writing the data in a specified blob field.

Delphi syntax:

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

C++ syntax:

```
virtual Classes::TStream* __fastcall CreateBlobStream(TField* Field,
TBlobStreamMode Mode);
```

Description

Call *CreateBlobStream* to obtain a stream for reading and writing the value of the field specified by the Field parameter. The Mode parameter indicates whether the stream will be used for reading the field's value (bmRead), writing the field's value (bmWrite), or modifying the field's value (bmReadWrite).

Blob streams are created in a specific mode for a specific field on a specific record. Applications create a new blob stream every time the record in the dataset changes: do not reuse an existing blob stream.

Note: It is preferable to call CreateBlobStream rather than creating a blob stream directly in code. This ensures that the stream is appropriate to the dataset, and may also ensure that datasets that do not always store BLOB data in memory fetch the blob data before creating the stream.

Note: With TurboDB you must free the stream object returned by *CreateBlobStream* before you

post the attached record. This is necessary because the stream does not write its content until it is destroyed.

1.2.3.16 TTdbDataSet.DatabaseName

Specifies the name of the database associated with this dataset.

Delphi syntax:

```
property DatabaseName: String;
```

C++ syntax:

```
__property AnsiString DatabaseName = {read=FDatabaseName, write
=SetDatabaseName};
```

Description

Use DatabaseName to specify the name of the database to associate with this dataset component. DatabaseName should match the name of a database component used in the application. You may also use a folder name like k:\turbodb\databases\db1 as the database name. In this case, all TurboDB tables in the folder are reachable via this *DatabaseName*. A third option is to set the file name of a single-file database as the database name.

Note: Attempting to set *DatabaseName* when a database already associated with this component is open raises an exception.

1.2.3.17 TTdbDataSet.FieldDefsTdb

Points to the list of field definitions for the dataset.

Delphi syntax:

```
property FieldDefsTdb: TTdbFieldDefs;
```

C++ syntax:

```
__property TTdbFieldDefs* FieldDefsTdb = {read=GetFieldDefsTdb, write
=SetFieldDefsTdb};
```

Description

FieldDefsTdb lists the field definitions for a dataset in a way specific to TurboDB. While an application can examine *FieldDefsTdb* to explore the field definitions for a dataset, it should not change these definitions unless creating a new table with *CreateTable*.

When you add, delete or modify field definitions in *FieldDefsTdb*, changes are reflected in the standard *TDataSet.FieldDefs* and vice versa. Use *FieldDefsTdb* to access the special fields and features of TurboDB like enumerations, links and relations. Use FieldDefs to work with *TTdbDataSet* in a way compatible to *TDataSet*.

To access fields and field values in a dataset, use the *Fields*, *AggFields*, and *FieldValues* properties, and the *FieldsByName* method.

1.2.3.18 TTdbDataSet.Filter

Contains an expression used for filtering the records of the data set.

Delphi syntax:

```
property Filter: string;
```

C++ syntax:

```
__property System::UnicodeString Filter;
```

Description

TurboDB supports two kinds of filters, conditional filters and keyword filters. Conditional filters work with a logical condition and show all records satisfying this condition. Conditions are described in Search-Conditions. The *FilterOptions* determine, how the filter is applied. With the *FilterMethod*

property you can determine, which mechanism is used for filtering.

TurboDB allows any valid TurboSQL search-condition as the filter expression here, not just the very limited set that is supported by the BDE components. The search-condition is interpreted according to the rules of VCL data sets:

- Floating numbers can be formatted either in the standard format with a decimal point or in the current local format (e.g. with a decimal comma).
- Time, date and timestamp values refer to the current local settings on the computer. You can use the native formats without quotes for time, date and timestamp literals to create locale independent filter expressions.
- The jokers for string comparisons with *like* are % and _ as in SQL.

In Delphi 2007 and before, you can use the *FilterW* property to set a Unicode filter expression.

The *TTdbTable* component allows also for keyword filters, which need a full-text index to work. Keyword filters are not assigned to the *Filter* property but to the *WordFilter* property.

See also

FilterOptions property
FilterMethod property

1.2.3.19 TTdbDataSet.Filtered

Specifies whether or not filtering is active for a dataset.

Delphi syntax:

```
property Filtered: Boolean;
```

C++ syntax:

```
__property WideString FilterW = {read=FFilterW, write=SetFilterW};
```

Description

Check *Filtered* to determine whether or not dataset filtering is in effect. If *Filtered* is True, then filtering is active. To apply filter conditions specified in the *Filter* property, in the *WordFilter* property or the *OnFilterRecord* event handler, set *Filtered* to True.

1.2.3.20 TTdbDataSet.FilterMethod

Specifies the filter method to use for next filter.

Delphi syntax:

```
property FilterMethod: TTdbFilterMethod;
```

C++ syntax:

```
__property TTdbFilterMethod FilterMethod;
```

Description

TurboDB supports incremental and static filtering. Incremental filtering means that the filter is applied to the records during browsing. Static filtering means that the result set of the filter is calculated and stored beforehand much like it is the case with queries. In most cases, static filters are faster and they have the additional advantage of delivering a stable result set with reliable record numbers that can be used for scroll bars etc. However in some special cases incremental filters are much faster then static ones on very large data sets (> 1 Million records):

- When the result is a large subset of the whole dataset, e.g. *Name <> 'Svyczkovski'.*
- When an index cannot be applied to speed searching, e.g. *Date > 2009-10-03 or Amount < 100000*.
- When only the first record satisfying the condition is requested.

- When a large result must additionally be sorted.

On the other hand, incremental filters have many disadvantages:

- The record count cannot be determined.
- There is no valid *RecNo* and therefore no scroll bar possible.
- They are very slow on huge data sets with only a few records satisfying the condition, e.g. *Name = 'Swyczkovski'*.

The *FilterMethod* property must be set before the *Filter* property is set. *FilterMethod* only operates on regular filters, not on word filters.

1.2.3.21 TTdbDataSet.FilterOptions

Specifies whether or not partial comparisons are permitted when filtering records.

Delphi syntax:

```
property FilterOptions: TFilterOptions;
```

C++ syntax:

```
__property TFilterOptions FilterOptions;
```

Description

Set *FilterOptions* to specify whether or not partial comparisons for matching filter conditions is allowed. The filter option value *foCaseInsensitive* is without function, because TurboDB works with table column collations that determine the case sensitivity.

When a string in a filter ends with an asterisk (*), it can be used to match partial strings. To disable matching of partial strings and to treat the asterisk as a literal character in string comparisons, set *FilterOptions* to include *foNoPartialCompare*.

1.2.3.22 TTdbDataSet.FilterW

Contains a Unicode expression used for filtering the records of the data set.

Delphi syntax:

```
property FilterW: WideString;
```

C++ syntax:

```
__property TTdbFieldDefs* FieldDefsTdb = {read=GetFieldDefsTdb, write
=SetFieldDefsTdb};
```

Description

FilterW is only needed in Delphi 2007 and below.

In those former versions of Delphi *Filter* used to be an *AnsiString* and could not accept non-ANSI characters. *FilterW* is the Unicode version of *Filter*. Use *FilterW* to define a filter expression containing non-ANSI characters. *FilterW* is not published because the Object Inspector in those former versions did not support Unicode characters. *Filter* and *FilterW* depend on each other. If you assign a value to *Filter*, *FilterW* will change accordingly and vice versa. The *FilterOptions* determine, how the filter is applied.

1.2.3.23 TTdbDataSet.GetEnumValue

Retrieves the numeric value for an enumeration constant.

Delphi syntax:

```
function GetEnumValue(FieldNo: Integer; const EnumStr: string): Integer;
```

C++ syntax:

```
int __fastcall GetEnumValue(int FieldNo, const AnsiString EnumStr);
```

Description

Call GetEnumValue to find out, which numeric value is assigned to a given enumeration constant. This function is used, if a TurboDB database table has a column of enumeration type.

1.2.3.24 TTdbDataSet.IntersectSelection

Intersects the current selection with a filter condition.

Delphi syntax:

```
procedure IntersectSelection(const Filter: UnicodeString);
```

C++ syntax:

```
int __fastcall IntersectSelection(const System::String Filter);
```

Description

IntersectSelection removes all those records from the current selection that do not satisfy the filter condition.

See also

Selections and Drill-Down

1.2.3.25 TTdbDataSet.IsSelected

Tests whether the current record is selected.

Delphi syntax:

```
function IsSelected(RecordNo: TTdbRecordNo): Boolean;
```

C++ syntax:

```
bool __fastcall IsSelected(TTdbRecordNo RecordNo);
```

Description

Use *IsSelected* if you want to know whether the current record is contained in the current selection.

See also

Selections and Drill-Down

1.2.3.26 TTdbDataSet.Locate

Searches a specified record and makes that record the active record.

Delphi syntax:

```
function Locate(const KeyFields: string; const KeyValues: Variant;
Options: TLocateOptions): Boolean;
```

C++ syntax:

```
virtual bool __fastcall Locate(const AnsiString KeyFields, const
System::Variant &KeyValues, TLocateOptions Options);
```

Description

Searches for a record in the dataset, where the fields identified by the semicolon-delimited string *KeyFields* have the values specified by the Variant or Variant array *KeyValues*. *Options* indicates whether the search is case insensitive and whether partial matches are supported. *Locate* returns True if a record is found that matches the specified criteria and the cursor repositioned to that record.

1.2.3.27 TTdbDataSet.Lookup

Retrieves field values from a record that matches specified search values.

Delphi syntax:

```
function Lookup(const KeyFields: String; const KeyValues: Variant; const
ResultFields: String): Variant;
```

C++ syntax:

```
virtual Variant __fastcall Lookup(const AnsiString KeyFields, const
Variant &KeyValues, const AnsiString ResultFields);
```

Description

Call Lookup to retrieve values for specified fields from a record that matches search criteria. *KeyFields* is a string containing a semicolon-delimited list of field names on which to search.

KeyValues is a variant array containing the values to match in the key fields. To specify multiple search values, pass *KeyValues* as a variant array as an argument, or construct a variant array on the fly using the *VarArrayOf* routine.

ResultFields is a string containing a semicolon-delimited list of field names whose values should be returned from the matching record.

Lookup returns a variant array containing the values from the fields specified in *ResultFields*. If the specified record could not be found, the variant is Null. If *ResultFields* contains only one item, the result value is simple (non-array) variant.

1.2.3.28 TTdbDataSet.OnProgress

Occurs during a time-consuming operation

Delphi syntax:

```
TTdbProgressEvent = procedure(Sender: TObject; PercentDone: Byte; var
Stop: Boolean) of object;
```

```
property OnProgress: TTdbProgressEvent;
```

C++ syntax:

```
typedef void __fastcall (__closure *TTdbProgressEvent)(System::TObject*
Sender, Byte PercentDone, bool &Stop);
```

```
__property TTdbProgressEvent OnProgress = {read=FOnProgress, write
=FOnProgress};
```

Description

Use *OnProgress* to update a progress bar and offer the user a way to cancel the operation. *OnProgress* is called during creating and altering a table and in the creation of indexes.

Note: Do not execute TurboDB database operations within the OnProgress event handler. Since the database engine is not reentrant calling TurboDB methods can lead to unpredictable results.

1.2.3.29 TTdbDataSet.OnResolveLink

Occurs when a link field has been changed to a value that cannot been resolved.

Delphi syntax:

```
TResolveLinkEvent = procedure(Sender: TObject; FieldNo: Integer; const
LinkInfo: string; var RecordId: Integer; var Cancel: Boolean);
```

```
property OnResolveLink: TResolveLinkEvent;
```

C++ syntax:

```
typedef void __fastcall (__closure *TResolveLinkEvent)(System::TObject*
Sender, int FieldNo, const AnsiString LinkInfo, int &RecordId, bool
```

```
&Cancel);
```

```
__property TResolveLinkEvent OnResolveLink = {read=FOnResolveLink, write
=FOnResolveLink};
```

Description

Write an *OnResolveLink* event handler to provide a custom procedure to find the record of the master table to link to. The handler is expected to set *RecordId* to the record id of the record in the master table that should be linked to the active record in the detail table. It can also set *Cancel* to True to prevent any linking from taking place.

1.2.3.30 TTdbDataSet.RecNo

Indicates the active record in the dataset.

Delphi syntax:

```
property RecNo: Integer;
```

C++ syntax:

```
__property int RecNo;
```

Description

If *IsSequenced* is *true*, *RecNo* returns the sequence number of the current record in the data set. It should not be used for any processing of the data but exclusively for scroll bars and other ways of indicating the current position within the data set to the user. If the amount of data is large, *RecNo* might return only an estimation of the position not the exact value.

1.2.3.31 TTdbDataSet.RemoveFromSelection

Removes records from the current selection.

Delphi syntax:

```
procedure RemoveFromSelection(RecordNo: TTdbRecordNo); overload;
```

```
procedure RemoveFromSelection(const Filter: UnicodeString); overload;
```

C++ syntax:

```
void __fastcall RemoveFromSelection(TTdbRecordNo RecordNo);
```

```
void __fastcall RemoveFromSelection(System::String Filter);
```

Description

Call the first overload to remove a specific record from the selection. Call the second overload to remove all records from the selection that comply to the filter condition.

See also

Selections and Drill-Down

1.2.3.32 TTdbDataSet.Replace

Replaces field values in a range of rows.

Delphi syntax:

```
procedure Replace(const Filter, Fields, Expressions: string): LongInt;
```

C++ syntax:

```
int __fastcall Replace(const AnsiString Filter, const AnsiString Fields,
const AnsiString Expressions);
```

Description

Replace sets the values of the *Fields* to the values calculated by *Expressions* for all records that meet the Filter's condition. The *Expressions* are evaluated in the context of each record.

Example

The following example appends the digit nine to all phone numbers from Reading, GB:

```
Replace('Phone like "+44 118*", 'Phone', 'Phone + "9"');
```

1.2.3.33 TTdbDataSet.SaveToFile

Stores the whole data set into a file.
Delphi syntax:

```
function SaveToFile(const FileName: string; Format: TTdbTableFormat):
Integer;
```

C++ syntax:

```
int __fastcall SaveToFile(const AnsiString FileName,
Tdbtypes::TTdbTableFormat Format);
```

Description

Use *SaveToFile* if you want to make a snapshot of your data set. *SaveToFile* internally uses the *TTdbBatchMove*component to create a file in one of the available formats containing all the data of the data set. *Format* defines the file type of the file.

This method is especially useful when you want to store the result set of a query for further processing. You may e.g. store the result set in TurboDB format and then open a *TTdbTable* component for it or run another query on it.

1.2.3.34 TTdbDataSet.Version

Indicates the version of the TurboDB component library and TurboDB Engine.

Delphi syntax:

```
property Version: String;
```

C++ syntax:

```
__property AnsiString Version = {read=GetVersion, write=SetVersion};
```

Description

Version is a string like 2.0.34/4.1.1. The part before the slash is the version of the TurboDB component library, the part behind indicates the TurboDB Engine version. Both version numbers have a main and a sub version number and a build number.

Note: Setting this property does not change its value.

1.2.3.35 TTdbForeignKeyAction

Indicates the way a table is protected or encrypted.

Unit

TdbDataSet

Delphi syntax:

```
type TTdbForeignKeyAction = (tiaReject, tiaSetNull, tiaSetDefault,
tiaCascade);
```

C++ syntax:

```
enum TTdbForeignKeyAction {tiaReject, tiaSetNull, tiaSetDefault,
tiaCascade};
```

Description

The values of this type describe, how TurboDB reacts on a violation of a foreign key constraint.

Value	Description

tiaReject	The modification, which leads to the constraint violation is not executed.
tiaSetNull	The foreign key fields in the dependent (child) table are set to null. Not yet implemented.
tiaSetDefault	The foreign key fields in the dependent (child) table are set to their default value. Not yet implemented.
tiaCascade	The corresponding rows in the dependent (child) table are deleted (if the parent row is deleted) or adjusted (if the parent row is adjusted).

1.2.3.36 TTdbForeignKeyDef

TTdbForeignKeyDef is used to define and indicate foreign key relationships from one table to another.

Unit

TdbDataSet

Description

TTdbForeignKeyDef defines a relationship between a child table and a parent table through corresponding field values.

1.2.3.37 TTdbForeignKeyDef Hierarchy

Hierarchy

TObject

|

TPersistent

|

TCollectionItem

|

TTdbForeignKeyDef

1.2.3.38 TTdbForeignKeyDef Methods

In TTdbForeignKeyDef

Assign

Derived from TCollectionItem

(check Embarcadero documentation for more information)

1.2.3.39 TTdbForeignKeyDef Properties

In TTdbTable

ChildFields

DeleteAction

Name

ParentTableName

ParentFields

UpdateAction

Derived from TCollectionItem
(check Embarcadero documentation for more information)

1.2.3.40 TTdbForeignKeyDef.Assign

Copies the properties of one foreign key definition to another.

Delphi Syntax

```
procedure Assign(Source: TPersistent); override;
```

C++ Syntax

```
virtual void __fastcall Assign(Classes::TPersistent* Source);
```

Description

Supports the standard VCL copying mechanism.

1.2.3.41 TTdbForeignKeyDef.ChildFields

Indicates the list of field names, which must are looked up in the parent table.

Delphi syntax:

```
property ChildFields: string;
```

C++ syntax:

```
__property AnsiString ChildFields = {read=GetChildFields, write
=SetChildFields};
```

Description

The values of the fields given here must correspond to the values of the *parent fields* in the *parent table*. Separate the column names by semicolon.

1.2.3.42 TTdbForeignKeyDef.DeleteAction

Defines the reaction of the database engine, when the parent row for a child row is deleted.

Delphi Syntax:

```
property DeleteAction: TTdbForeignKeyAction;
```

C++ Syntax:

```
__property TTdbForeignKeyAction DeleteAction = {read=GetDeleteAction,
write=SetDeleteAction};
```

Description

When a row in the parent table is deleted, the action defined will be applied to all rows in the child table, which relate to the delete row in the parent table.

See also

TTdbForeignKeyAction

1.2.3.43 TTdbForeignKeyDef.Name

Indicates the name of the foreign key constraint.

Delphi Syntax:

```
property Name: string
```

C++ Syntax:

```
__property AnsiString Name = {read=GetName, write=SetName};
```

Description

The name is used to identify the foreign key constraint. It can also be referred to in SQL statements.

1.2.3.44 TTdbForeignKeyDef.ParentTableName

Indicates the parent table for this foreign key constraint.

Delphi Syntax:

```
property ParentTableName: string;
```

C++ Syntax:

```
__property AnsiString ParentTableName = {read=GetParentTableName, write
=SetParentTableName};
```

Description

ParentTableName must identify another table within the same database, to which the child table is related. The values of the child fields of each row in the child table must correspond to the values of the parent fields in one row in the parent table.

1.2.3.45 TTdbForeignKeyDef.ParentFields

Indicates the list of columns in the parent table.

Delphi Syntax:

```
property ParentFields: string;
```

C++ Syntax:

```
__property AnsiString ParentFields = {read=GetParentFields, write=Set
ParentFields};
```

Description

There must be as many column names (separated by semicolon) in this property as are in the ChildFields property. There must exist a row in the parent table for each row in the child table, where the values of these fields are the same as the values of the child fields in the child table row.

1.2.3.46 TTdbForeignKeyDef.UpdateAction

Defines the reaction of the database engine, when the parent row for a child row is modified.

Delphi Syntax:

```
property UpdateAction: TTdbForeignKeyAction;
```

C++ Syntax:

```
__property TTdbForeignKeyAction UpdateAction = {read=GetUpdateAction,
write=SetUpdateAction};
```

Description

When a row in the parent table is modified in a way that it does no more fit the values in the corresponding child table rows, the action defined will be applied to all those child table rows.

See also

TTdbForeignKeyAction

1.2.3.47 TTdbForeignKeyDefs

TTdbForeignKeyDefs stores all foreign keys of a table.

Unit

TdbDataSet

Description

TTdbForeignKeyDefs contains all *TTdbForeignKeyDef* objects that belong to a table. The property *TTdbTable.ForeignKeyDefs* contains a *TTdbForeignKeyDefs* objects.

1.2.3.48 TTdbForeignKeyDefs Hierarchy

Hierarchy

TObject

|

TPersistent

|

TCollection

|

TOwnedCollection

|

TDefCollection

|

TTdbForeignKeyDefs

1.2.3.49 TTdbForeignKeyDefs Methods

In *TTdbForeignKeyDefs*

Add

Derived from *TDefCollection*

(check Embarcadero documentation for more information)

1.2.3.50 TTdbForeignKeyDefs.Add

Creates a new foreign key and adds it to the collection.

Delphi Syntax

```
function Add(ParentTableName, ParentFields, ChildFields: string;
UpdateAction: TTdbForeignKeyAction = tiaReject; DeleteAction:
TTdbForeignKeyAction = tiaReject): TTdbForeignKeyDef; overload;
```

C++ Syntax

```
virtual TTdbForeignKeyDef* __fastcall Add(const System::String
ParentTableName, const System::String ParentFields, const System::String
ChildFields, TTdbForeignKeyAction UpdateAction = tiaReject,
TTdbForeignKeyAction DeleteAction = tiaReject);
```

Description

Use *Add* to create a foreign key for a table within one function call.

1.2.3.51 TTdbFulltextIndexDef

TTdbFulltextIndexDef describes a full-text index in a database table.

Unit

TdbDataSet

Description

Use the properties and methods of a full-text index definition to:

- Create a full-text index in a table.
- Display and check existing full-text indexes.
- Identify the fields, that make up the full-text index.
- Determine the name of a full-text index.

TTdbFullTextIndexDef applies to the new type of full-text index only, which is available for table level 4 and above.

1.2.3.52 TTdbFullTextIndexDef Hierarchy

Hierarchy

TObject

|

TPersistent

|

TCollectionItem

|

TNamedItem

|

TTdbFullTextIndexDef

1.2.3.53 TTdbFullTextIndexDef Methods

In TTdbTable

Assign

Derived from TCollectionItem

(check Embarcadero documentation for more information)

1.2.3.54 TTdbFullTextIndexDef Properties

In TTdbTable

Dictionary

Fields

MinRelevance

Options

Derived from TNamedItem

(check Embarcadero documentation for more information)

1.2.3.55 TTdbFullTextIndexDef.Assign

Copies the properties of one full-text index definition to another.

Delphi Syntax

```
procedure Assign(Source: TPersistent); override;
```

C++ Syntax

```
virtual void __fastcall Assign(Classes::TPersistent* Source);
```

Description

Supports the standard VCL copying mechanism.

1.2.3.56 TTdbFulltextIndexDef.Dictionary

Indicates the name of the dictionary table for the full-text index.

Delphi Syntax:

```
property Dictionary: string;
```

C++ Syntax:

```
__property AnsiString Dictionary = {read=GetDictionary, write
=SetDictionary};
```

Description

A full-text index always refers to a dictionary database table, which contains the words that have been/can be indexed. This table must exist before the full-text index can be created.

See also

Creating a Full-Text Index at Design-Time, Creating a Full-Text Index at Run-Time

1.2.3.57 TTdbFulltextIndexDef.Fields

Indicates the fields to index in the full-text index.

Delphi Syntax:

```
property Fields: string;
```

C++ Syntax:

```
__property AnsiString Fields = {read=GetFields, write=SetFields};
```

Description

Fields is a semicolon-separated list of field names in the table, which are included in the full-text index.

1.2.3.58 TTdbFulltextIndexDef.MinRelevance

Defines the minimum relevance for words to be indexed.

Delphi Syntax:

```
property MinRelevance: SmallInt;
```

C++ Syntax:

```
__property short MinRelevance = {read=GetMinRelevance, write
=SetMinRelevance};
```

Description

Words with a lower relevance than defined with this property are not be indexed, this means one cannot search for such a word. Relevance is a positive number between 0 (absolutely not relevant) to 100 (highly relevant). Relevance support is not yet fully implemented.

1.2.3.59 TTdbFulltextIndexDef.Options

Describes the characteristics of the full-text index.

Delphi Syntax

```
property Options: TFulltextIndexOptions;
```

C++ Syntax:

```
__property TFulltextIndexOptions Options = {read=FOptions, write
=SetOptions, nodefault}
```

Description

When creating a new full-text index, use *Options* to specify the attributes of the index. *Options* can contain zero or more of the *TFullltextIndexOption* constants.

When inspecting the definitions of existing full-text indexes, read *Options* to determine the option (s) used to create the index.

1.2.3.60 TTdbFulltextIndexOptions

TTdbFullltextIndexOptions describes the attributes of an index.

Unit

TdbDataSet

Delphi syntax:

```
type
  TTdbFulltextIndexOption = (tfoUpdateDictionary);
  TTdbFulltextIndexOptions = set of TTdbFulltextIndexOption;
```

C++ syntax:

```
enum TFulltextIndexOption {tfoUpdateDictionary};
typedef Set<TIndexOption, tfoUpdateDictionary, tfoUpdateDictionary>
TTdbFulltextIndexOptions;
```

Description

TTdbFullltextIndexOptions is a set of attributes that applies to a specific full-text index. A *TTdbFullltextIndexOptions* value can include zero or more of the following values:

Value	Description
`tfoUpdateDictionary`	During indexing, TurboDB will add words found in the table but not in the dictionary to the dictionary.

1.2.3.61 TTdbTable

TTdbTable encapsulates a TurboDB database table.

Unit

TdbDataSet

Description

Use *TTdbTable* to access data in a single TurboDB table. *TTdbTable* provides direct access to every record in the table. A table component can also work with a subset of records within a database table using filters.

1.2.3.62 TTdbTable Hierarchy

Hierarchy

TObject

|

TPersistent

|

TComponent

|

TDataSet

|

TTdbDataSet

|

TTdbTable

1.2.3.63 TTdbTable Methods

In TTdbTable

AddIndex
AlterTable
BatchMove
CreateTable
DeleteIndex
DeleteTable
EditKey
EmptyTable
Exists
FindKey
FindNearest
GetUsage
GotoKey
GotoNearest
LockTable
RenameTable
SetKey
UnlockTable
UpdateFullTextIndex
UpdateIndex

Derived from TTdbDataSet

GetEnumValue
Locate
Lookup
Replace

Derived from TDataSet

(check Embarcadero documentation for more information)

1.2.3.64 TTdbTable Properties

In TTdbTable

DetailFields
Exclusive
FlushMode
FullTextTable
IndexDefs
IndexName
Key
LangDriver
MasterSource
Password
ReadOnly
TableFileName
TableLevel
TableName

Derived from TTdbDataSet

DatabaseName
FieldDefsTdb
Filter
Filtered
FullTextTable
Version

Derived from TDataSet

(check Embarcadero documentation for more information)

1.2.3.65 TTdbTable Events

Derived from TTdbDataSet

OnProgress
OnResolveLink

Derived from TDataSet

(check Embarcadero documentation for more information)

1.2.3.66 TTdbTable.AddFulltextIndex

Creates a new full-text index for the table.

Delphi syntax:

```
procedure AddFulltextIndex(const Fields, RelationField,
CounterIndexFileName: string; Limit: Integer);
```

C++ syntax:

```
void __fastcall AddFulltextIndex(const AnsiString Fields, const
AnsiString RelationField, const AnsiString CounterIndexFileName, int
Limit);
```

Description

Call *AddFulltextIndex* to create a full-text index for the table. This method creates the classic form of a non-updatable full-text index.

Fields is the list of column names to include in the full-text index. *RelationField* is the relation field in the table, that will be used to connect the dictionary table with the table. *CounterIndexFileName* is the name of a text-file containing words, that should not be included in the index. Limit is the number of occurrences per word, after which it will be removed from the index.

Remarks

It is recommended to upgrade the table to level 4 or higher and use AddFulltextIndex2 to create an updatable full-text index.

See also

AddFulltextIndex2

1.2.3.67 TTdbTable.AddFulltextIndex2

Create a new 2nd generation full-text index for the table.

Delphi Syntax:

```
procedure TTdbTable.AddFulltextIndex2(
     const Name, Fields, Dictionary: AnsiString;
     const Separators: UnicodeString = ''
);
```

C++ Syntax:

```
void __fastcall AddFulltextIndex2(
     const AnsiString Name,
     const AnsiString Fields,
     const AnsiString Dictionary,
     const UnicodeString Separators = ""
);
```

Description

This method is available for table level 4 and above only. It adds a new updatable full-text index to the table. *Name* is the name of the index, *Fields* a comma-separated list of fields to include in the index and *Dictionary* the name of the full-text index table containing all the words.

Separators is a list of separating characters when the text in the table or a search-condition are broken down into words. If an empty string is passed, a default separator list is used:

```
 °^!?'"§$%&/\()[]{}<>=´`+-*~'#,.;:@|
```

If a non-empty string is passed, all characters with an encoding < 32 plus the given characters are regarded as separators.

Compatibility

User-defined separators are available only in table level 6 and above. Trying to specify them for a table of a lower level will result in an "Unsupported table feature" exception.

See also

CREATE FULLTEXTINDEX TurboSQL command

1.2.3.68 TTdbTable.AddIndex

Creates a new index for the table.

Delphi syntax:

```
procedure AddIndex(const Name, Fields: String; Options: TIndexOptions,
const DescFields: String = '');
```

C++ syntax:

```
void __fastcall AddIndex(const AnsiString Name, const AnsiString Fields,
Db::TIndexOptions Options, const AnsiString DescFields = "");
```

Description

Call *AddIndex* to create an index the table. An index can be used to show the records in a given order or to search faster for certain records.

Name is the name of the new index.

Fields is the list of tables column to use for the index separated by semi-colons.

Options is a set of index options. The options applicable for TurboDB indexes are:

ixExpression	Fields contains an TurboDB expression to calculate the index entries
ixUnique	Each index entry can occur only once

DescFields is a list of fields in *Fields* that are to be sorted in descending order.

Remarks

This method is equivalent to *TTable.AddIndex*.

1.2.3.69 TTdbTable.AlterTable

Restructures the database table.

Delphi syntax:

```
procedure AlterTable;
```

C++ syntax:

```
void __fastcall AlterTable(void);
```

Description

Use AlterTable to modify the columns, the table level, the password or key of a database table. Before you call *AlterTable* set the *FieldDefsTdb*, the *TableLevel*, the EncryptionMethod, the Password and the Capacity properties to the desired values. If the OnProgress event is assigned, AlterTable notifies the handler during the altering process.

Note: Do not call any TurboDB database methods in the OnProgress event handler during altering a table since this can leave the TurboDB Engine in a indeterminate state.

1.2.3.70 TTdbTable.BatchMove

Moves records from a dataset into this table.

Delphi Syntax:

```
function BatchMove(ASource: TDataSet; AMode: TTdbBatchMode): LongInt;
```

C++ syntax:

```
int __fastcall BatchMove(Db::TDataSet* ASource, Tdbtypes::TTdbBatchMode
```

```
AMode);
```

Description

Call BatchMove to

- Copy records from another table into this table.
- Update records in this table that occur in another table.
- Append records from another table to the end of this table.
- Delete records in this table that occur in another table.

ASource is a dataset component containing the records to import or (if deleting) match. The *AMode* parameter indicates what operation to perform (copy, update, append, or delete). This table is the destination of the batch operation. *BatchMove* returns the number of records operated on.

1.2.3.71 TTdbTable.Capacity

Indicates the number of records a table must be able to store at least.

Delphi syntax:

```
property Capacity: LongInt;
```

C++ syntax:

```
__property long Capacity = {read, write, nodefault};
```

Description

While all database tables in TurboDB can hold up to 2 billion records, indexes are limited depending on the key size and page size. Set this property before creating or altering a table to make sure subsequent indexes will be created with the correct settings. There is no need to set this property correctly, an order of magnitude is sufficient. If you think your table will contain up to 100,000 records, set this property to 200,000 etc. Indexes for this will then have a capacity of at least the given number but mostly larger. (The real capacity of an index can be seen in TurboDB Viewer.) The purpose of this property is therefore mainly one of optimization.

Note

This property can be set only for tables of level 4 and higher. When creating indexes for older tables, the capacity is taken from the property *IndexCapacity* in the Database component.

1.2.3.72 TTdbTable.Collation

Indicates and sets the collation of this table.

Delphi syntax:

```
property Collation: string;
```

C++ syntax:

```
__property AnsiString Collation;
```

Description

The collation of the table is the default collation for all textual columns in the table (see *TTdbFieldDef.Specification*). This property is used for creating and altering tables. If you set this value to an empty string, the collation *TurboDB* is used.

Collation replaces the *LangDriver* property used in TurboDB 5 and below.

See also

Collations
TTdbFieldDef.Specification

1.2.3.73 TTdbTable.CreateTable

Creates a new database table.

Delphi syntax:

```
procedure CreateTable;
```

C++ syntax:

```
void __fastcall CreateTable(void);
```

Description

Call *CreateTable* to create a new database table. Set the FieldDefsTdb, DatabaseName, TableName, TableLevel, EncryptionMethod, Password and Capacityproperties to the desired values before you execute this function. If the *OnProgress* event is assigned, *CreateTable* notifies the handler during the altering process.

Note

Do not call any TurboDB database methods in the OnProgress event handler during table creation since this can leave the TurboDB Engine in a indeterminate state.

1.2.3.74 TTdbTable.DeleteAll

Deletes all records within the current filter range.

Delphi syntax:

```
procedure DeleteAll;
```

C++ syntax:

```
void __fastcall DeleteAll(void);
```

Description

Use *DeleteAll* to delete a bunch of records. If a filter is active, all filtered records are deleted. If no filter is active, all records of the tables are deleted. In this case DeleteAll works similar to EmptyTable, however it is slower but does not need exclusive access to the table.

1.2.3.75 TTdbTable.DeleteIndex

Deletes an index of the table.

Delphi syntax:

```
procedure DeleteIndex(const Name: string);
```

C++ syntax:

```
void __fastcall DeleteIndex(const AnsiString Name);
```

Description

Call *DeleteIndex* to remove an index from the table and erase its file.

Name is the file name of the index including the extension. It is identical to the corresponding item in IndexFiles.

Remark

TTdbTable.DeleteIndex is equivalent to *TTable.DeleteIndex*.

1.2.3.76 TTdbTable.DeleteTable

Deletes a database table.

Delphi syntax:

```
procedure DeleteTable;
```

C++ syntax:

```
void __fastcall DeleteTable(void);
```

Description

Use *DeleteTable* to erase the database table and all corresponding files.

Note: The data contained in the table cannot be recovered.

1.2.3.77 TTdbTable.DetailFields

Defines the field names used to link the table to a master data source.

Delphi syntax:

```
property DetailFields: string;
```

C++ syntax:

```
__property AnsiString DetailFields = {read=FDetailFields, write
=SetDetailFields};
```

Description

Set *DetailFields* to indicate the fields whose values must equal the corresponding fields of the master table. *DetailFields* is a string containing one or more field names in the detail table. Separate field names with semicolons. When you use DetailFields you must also set MasterFields to the corresponding field names of the master table. When you set *MasterFields* but not *DetailFields*, TurboDB expects the field names of the detail table to match those of the master table.

1.2.3.78 TTdbTable.EditKey

Enables modification of the search key buffer.

Delphi syntax:

```
procedure EditKey;
```

C++ syntax:

```
void __fastcall EditKey(void);
```

Description

Call *EditKey* to put the dataset in dsSetKey state while preserving the current contents of the current search key buffer. To determine current search keys, you can use the IndexFields property to iterate over the fields used by the current index.

EditKey is especially useful when performing multiple searches where only one or two field values among many change between each search.

1.2.3.79 TTdbTable.EmptyTable

Clears all records in the table.

Delphi syntax:

```
procedure EmptyTable;
```

C++ syntax:

```
void __fastcall EmptyTable(void);
```

Description

Use *EmptyTable* to delete all records from the table.

Note: The records deleted cannot be recovered.

Remark: TTdbTable.EmptyTable is equivalent to TTable.EmptyTable.

1.2.3.80 TTdbTable.EncryptionMethod

Defines whether and how a table is protected or encrypted.

Delphi syntax:

```
property EncryptionMethod: TTdbEncryptionMethod
```

C++ syntax:

```
__property TTdbEncryptionMethod EncryptionMethod = {read, write,
nodefault};
```

Description

A TurboDB table can be protected and encrypted in different ways. This property indicates which method has been chosen and is used to determine the encryption method when creating or altering a table with the *CreateTable* or *AlterTable* method.

See also

TTdbEncryptionMethod, Password, Data Security

1.2.3.81 TTdbTable.Exclusive

Enables an application to gain sole access to a TurboDB table.

Delphi syntax:

```
property Exclusive: Boolean;
```

C++ syntax:

```
__property bool Exclusive = {read=FExclusive, write=SetExclusive,
nodefault};
```

Description

Use *Exclusive* to prevent other applications from accessing a TurboDB table while it is open in this application. Before opening the table, set *Exclusive* to True. A table must be closed before changing the *Exclusive* property.

When *Exclusive* is True, then when the application successfully opens the table, no other application can access it. If the table for which the application has requested exclusive access is already in use by another application, an exception is raised. To handle such exceptions, wrap the code that opens the table in a **try**..**except** block.

Do not set *Exclusive* to True at design time if you also set the *Active* property to True at design time. In this case an exception is raised at start-up because the table is already in use by the IDE.

Warning

Exclusive access to a table within a singlefile database is not possible and will raise an exception when trying.

Open the whole database in exclusive mode instead.

1.2.3.82 TTdbTable.Exists

Indicates whether the underlying database table exists.

Delphi syntax:

```
property Exists: Boolean;
```

C++ syntax:

```
__property bool Exists = {read=GetExists, nodefault};
```

Description

Read *Exists* at runtime to determine whether a database table exists. If the table does not exist, create a table from the field definitions and index definitions using the *CreateTable* method. This property is read-only.

1.2.3.83 TTdbTable.FindKey

Searches for a record containing specified field values.

Delphi syntax:

```
function FindKey(const KeyValues: array of const): Boolean;
```

C++ syntax:

```
bool __fastcall FindKey(const System::TVarRec* KeyValues, const int
KeyValues_Size);
```

Description

Call *FindKey* to search for a specific record in a dataset. *KeyValues* contains a comma-delimited array of field values, called a key. Each value in the key can be a literal, a variable, a null, or nil. If the number of values passed in *KeyValues* is less than the number of columns in the index used for the search, the missing values are assumed to be null.

The key must always be an index, which can be specified in the *IndexName* property. If *IndexName* is empty, *FindKey* uses the table's primary index.

If the search is successful, *FindKey* positions the cursor on the matching record and returns True. Otherwise the cursor is not moved, and *FindKey* returns False.

1.2.3.84 TTdbTable.FindNearest

Moves the cursor to the record that most closely matches a specified set of key values.

Delphi syntax:

```
procedure FindNearest(const KeyValues: array of const);
```

C++ syntax:

```
void __fastcall FindNearest(const System::TVarRec* KeyValues, const int
KeyValues_Size);
```

Description

Call *FindNearest* to move the cursor to a specific record in a dataset or to the first record in the dataset that is greater than the values specified in the KeyValues parameter. KeyValues contains a comma-delimited array of field values, called a key. Each value in the key can be a literal, a variable, a null, or nil. If the number of values passed in KeyValues is less than the number of columns in the index used for the search, the missing values are assumed to be null.

The key must always be an index, which can be specified in the *IndexName* property. If *IndexName* is empty, *FindNearest* uses the table's primary index.

FindNearest positions the cursor either on a record that exactly matches the search criteria, or on the first record whose values are greater than those specified in the search criteria.

1.2.3.85 TTdbTable.FlushMode

Switches write buffering on and off.

Delphi syntax:

```
property FlushMode: TTdbFlushMode;
```

C++ syntax:

```
__property TTdbFlushMode FlushMode = {read=FFlushMode, write=FFlushMode,
nodefault};
```

Description

If *FlushMode* is set to *tfmSecure*, records are written to disk and the directory information is updated immediately. This mode minimizes the data loss in case of a current supply breakdown or a crash of the application. To optimize write performance of your application, set FlushMode to *tfmFast* and such enable write buffering. *tfmDefault* uses the *FlushMode* value of the database component.

1.2.3.86 TTdbTable.ForeignKeyDefs

Contains information about the foreign key constraints for a table.

Delphi Syntax:

```
property ForeignKeyDefs: TTdbForeignKeyDefs;
```

C++ Syntax:

```
__property TdbDataSet::TTdbForeignKeyDefs* ForeignKeyDefs =
{read=FForeignKeyDefs, write=SetForeignKeyDefs, stored=FStoreDefs};
```

Description

ForeignKeyDefs is a collection of foreign key definitions, each of which describes one foreign key constraint for the table. Define the foreign key definitions of a table before calling *CreateTable* or *AlterTable*.

If *ForeignKeyDefs* is updated or manually edited, the *StoreDefs* property becomes true.

Note: The information in *ForeignKeyDefs* may not always reflect the current keys available for a table. Before examining *FulltextIndexDefs*, call its *Update* method to refresh the list.

1.2.3.87 TTdbTable.FulltextIndexDefs

Contains information about the indexes for a table.

Delphi syntax:

```
property FulltextIndexDefs: TTdbFulltextIndexDefs;
```

C++ syntax:

```
__property TdbDataSet::TTdbFulltextIndexDefs* FulltextIndexDefs =
{read=FFullltextIndexDefs, write=SetFulltextIndexDefs,
stored=FStoreDefs};
```

Description

FulltextIndexDefs is a collection of full-text index definitions, each of which describes an available full-text index for the table. Define the index definitions of a table before calling CreateTable or creating a table at design time.

If *FulltextIndexDefs* is updated or manually edited, the *StoreDefs* property becomes true.

Note: The index definitions in *FulltextIndexDefs* may not always reflect the current indexes available for a table. Before examining *FulltextIndexDefs*, call its *Update* method to refresh the list.

1.2.3.88 TTdbTable.FullTextTable

Indicates the TurboDB table used as an index for full-text search.

Delphi syntax:

```
property FullTextTable: TTdbTable;
```

C++ syntax:

```
__property TTdbTable* FullTextTable = {read=FFullTextTable, write
=FFullTextTable};
```

Description

Set *FullTextTable* to a table that contains the keywords for a keyword filter. *FullTextTable* must be

linked to the table you are searching in by a relation field in the base table. In order to filter for keywords WordFilter must contain a valid keyword search expression.

1.2.3.89 TTdbTable.GetIndexNames

Retrieves a list of available indexes for a table.

Delphi syntax:

```
procedure GetIndexNames(List: TStrings);
```

C++ syntax:

```
void __fastcall GetIndexNames(Classes::TStrings* List);
```

Description

Call *GetIndexNames* to retrieve a list of all available indexes for a table. List is a string list object, created and maintained by the application, into which to retrieve the index names.

1.2.3.90 TTdbTable.GetUsage Method

Retrieves information about the current usage of the table by all accessing applications.

Delphi syntax:

```
procedure GetUsage(out TableUsage: TTdbTableUsage);
```

C++ syntax:

```
void __fastcall GetUsage(TTdbTableUsage& TableUsage);
```

Description

This function reports information about the applications that currently access this table and their lock status.

See also

TTdbTableUsage type

1.2.3.91 TTdbTable.GotoKey

Moves the cursor to a record specified by the current key.

Delphi syntax:

```
function GotoKey: Boolean;
```

C++ syntax:

```
bool __fastcall GotoKey(void);
```

Description

Use *GotoKey* to move to a record specified by key values assigned with previous calls to *SetKey* or *EditKey* and actual search values indicated in the Fields property.

If *GotoKey* finds a matching record, it positions the cursor on the record and returns True. Otherwise the current cursor position remains unchanged, and *GotoKey* returns False.

1.2.3.92 TTdbTable.GotoNearest

Moves the cursor to the record that most closely matches the current key.

Delphi syntax:

```
procedure GotoNearest;
```

C++ syntax:

```
void __fastcall GotoNearest(void);
```

Description

Call *GotoNearest* to position the cursor on the record that is either the exact record specified by the current key values in the key buffer, or on the first record whose values exceed those specified.

Before calling *GotoNearest*, an application must specify key values by calling *SetKey* or *EditKey* to put the dataset is *dsSetKey* state, and then use *FieldByName* to populate the key buffer property with search values.

1.2.3.93 TTdbTable.IndexDefs

Contains information about the indexes for a table.

Delphi syntax:

```
property IndexDefs: TIndexDefs;
```

C++ syntax:

```
__property Db::TIndexDefs* IndexDefs = {read=FIndexDefs, write
=SetIndexDefs, stored=IndexDefsStored};
```

Description

IndexDefs is a collection of index definitions, each of which describes an available index for the table.

Note: The index definitions in *IndexDefs* may not always reflect the current indexes available for a table. Before examining *IndexDefs*, call its Update method to refresh the list.

1.2.3.94 TTdbTable.IndexName

Indicates the name of the currently used index.

Delphi syntax

```
property IndexName: String;
```

C++ syntax

```
__property AnsiString IndexName = {read=FIndexName, write=SetIndexName};
```

Description

Set *IndexName* to show the records of the table in the order defined by the index. Use one of items in the IndexFiles property. Setting *IndexName* to an empty string shows the records in the natural order, that is the order the records are stored in the table file.

1.2.3.95 TTdbTable.Key

Contains the key for decrypting an encrypted table.

This property is no more supported since TurboDB 5. When upgrading an application from version 4 and below, please observe the upgrade notes.

1.2.3.96 TTdbTable.LangDriver

Obsolete, do not use anymore. See collations and Collation.

Indicates the language driver used for this table.

Delphi syntax:

```
property LangDriver: String;
```

C++ syntax:

```
__property AnsiString LangDriver = {read=FLangDriver, write
=FLangDriver};
```

Description

Check this property to learn the language the database table is bound to. The value of this property is identical to the extension of the language driver dll used with this table. E.g. if *LangDriver* is set to *fra*, the table uses the dynamic link library *TdbLanDr.fra* for string comparisons.

Set this property to determine the language for a table in a subsequent call to *CreateTable* or *AlterTable*. The language driver must exist at the time of creating or altering the table.

1.2.3.97 TTdbTable.LockTable

Locks a TurboDB table.

Delphi syntax:

```
procedure LockTable(LockType: TTdbLockType);
```

C++ syntax:

```
void __fastcall LockTable(TTdbLockType LockType);
```

Description

Call *LockTable* to lock a database table to prevent other applications from placing a particular type of lock on the table. LockType specifies the lock requested by this application.

Requesting a write lock prevents other applications from writing to a table. Requesting a total lock prevents other application from writing to and reading a table.

1.2.3.98 TTdbTable.MasterFields

Specifies one or more fields in a master table to link with corresponding fields in this table in order to establish a master-detail relationship between the tables.

Delphi syntax:

```
property MasterFields: String;
```

C++ syntax:

```
__property AnsiString MasterFields = {read=GetMasterFields, write
=SetMasterFields};
```

Description

Use *MasterFields* after setting the *MasterSource* property to specify the names of one or more fields to use in establishing a detail-master relationship between this table and the one specified in *MasterSource*.

MasterFields is a string containing one or more field names in the master table. Separate field names with semicolons.

Each time the current record in the master table changes, the new values in those fields are used to select corresponding records in this table for display. You can set DetailFields to define the fields of the detail table that must match the field values of the master table.

If you set the *MasterSource* property but not the *MasterFields* and the *DetailFields* property, TurboDB uses the default relationship between the master table and this table to establish the master-detail relationship. The default relationship is defined by the link and relation fields in both tables.

1.2.3.99 TTdbTable.MasterSource

References the master data source for a master-detail view.

Delphi syntax:

```
property MasterSource: TDataSource;
```

C++ syntax:

```
__property Db::TDataSource* MasterSource = {read=FMasterSource, write
=SetDataSource};
```

Description

Use *MasterSource* to specify the name of the data source component whose *DataSet* property identifies a data set to use as a master table in establishing a master-detail relationship between this table and another one. Each time the current record in the master table changes, the new values in those fields are used to select corresponding records in this table for display.

TurboDB can link the records of the detail table to the records in the master table in three different ways:

- The default relationship is defined by link and relations fields within the tables. To use this kind of relationship, leave the *MasterFields* and the *DetailFields* properties empty.
- The standard relationship is defined by the *MasterFields* property and works only when the matching fields in the detail and the master table have the same names.
- To match detail fields and master fields with different field names, you can use the *DetailFields* property in addition to the *MasterFields* property.

Note: At design time choose an available data source from the *MasterSource* property's drop-down list in the Object Inspector.

Attention: All tables used to establish a master-detail relationship must belog to the same database

1.2.3.100 TTdbTable.Password

Specifies the password used to open or create the table.

Delphi syntax:

```
property Password: string;
```

C++ syntax:

```
__property AnsiString Password = {read=FPassword, write=FPassword};
```

Description

Set *Password* before opening a protected table. If *Password* has the wrong value, the database component issues an OnPassword event. If there is no event handler defined, the table won't open and an *ETurboDBError* exception is raised.

When you create or alter a table the current values of *EncryptionMethod* and *Password* is used to protect the table.

If the encryption method of the table is *temClassic*, an alphanumeric key is needed as well as a 32 bit number. In this case set the password property to the table password concatenated with the number using a semicolon, e.g. *MyPass;-7896879*. It is however recommended to change such tables to encryption mode *fast encrypt* for easier handling of the password.

See also

EncryptionMethod

1.2.3.101 TTdbTable.ReadOnly

Indicates, if clients can modify the data set.

Delphi syntax:

```
property ReadOnly: Boolean;
```

C++ syntax:

```
__property bool ReadOnly = {read=FReadOnly, write=SetReadOnly,
nodefault};
```

Description

Set *ReadOnly* to True, if you want to open the data set in read-only mode. Calling a method that would modify the data set will then raise an exception.

1.2.3.102 TTdbTable.RenameTable

Renames a TurboDB database table and all joined files.

Delphi syntax:

```
procedure RenameTable(const NewTableName: String);
```

C++ syntax:

```
void __fastcall RenameTable(const AnsiString NewTableName);
```

Description

Use *RenameTable* if you want to give another name to the table. You must close the table before you can rename it. This method renames the table file, the default index files and the memo and blob files. Apply this method carefully since all applications using this database table are affected.

1.2.3.103 TTdbTable.RepairTable

Repairs a database table.

Delphi Syntax:

```
procedure RepairTable;
```

C++ Syntax:

```
void __fastcall RepairTable();
```

Description

Use *RepairTable* if the database table contains faults like incorrect memo or blob links. The table will be re-built and the errors will be corrected where possible. You should create a backup copy before the call to *RepairTable*.

1.2.3.104 TTdbTable.SetNextAutoIncValue

Sets the starting point for future *AutoInc* values.

Delphi syntax:

```
procedure SetNextAutoIncValue(FieldNo: SmallInt; Value: LongInt);
```

C++ syntax:

```
void __fastcall SetNextAutoIncValue(SmallInt FieldNo, LongInt Value);
```

Description

Use this function to define, which values are created for auto-increment fields. Be very careful as setting the next auto-increment to a lower value will result in duplicate auto-increment values in the table. This method is especially helpful after executing a batch move with the *RecalcAutoInc* property set to false.

1.2.3.105 TTdbTable.SetKey

Enables setting of keys and ranges for a dataset prior to a search.

Delphi syntax:

```
procedure SetKey;
```

C++ syntax:

```
void __fastcall SetKey(void);
```

Description

Call *SetKey* to put the dataset into *dsSetKey* state and clear the current contents of the key buffer. The FieldByName method can then be used to supply a new set of search values prior to conducting a search.

Note: To modify an existing key or range, call *EditKey*.

1.2.3.106 TTdbTable.TableFileName

Indicates the whole path name of the underlying database table file.

Delphi syntax:

```
property TableFileName: String;
```

C++ syntax:

```
__property AnsiString TableFileName = {read=GetTableFileName};
```

Description

TableFileName is a read-only property that shows the file name of the database table including drive (on Windows) and directory.

1.2.3.107 TTdbTable.TableLevel

Indicates the version of the database table files.

Delphi syntax:

```
property TableLevel: Integer;
```

C++ syntax:

```
__property int TableLevel = {read=FTableLevel, write=SetTableLevel,
nodefault};
```

Description

Read *TableLevel* to determine the version of the database table file. Some features of the table depend on the TableLevel, e.g. DateTime fields are available only from TableLevel 3 on. When you create or alter a database table at run-time you must set the TableLevel to the value you need. A description for the table levels is found in "Table Levels".

1.2.3.108 TTdbTable.TableName

Indicates the name of the database table that this component encapsulates.

Delphi syntax:

```
property TableName: TFileName;
```

C++ syntax:

```
__property AnsiString TableName = {read=FTableName, write=SetTableName};
```

Description

Use *TableName* to specify the name of the database table this component encapsulates. To set *TableName* to a meaningful value, the *DatabaseName* property should already be set.

Note: To set *TableName*, the *Active* property must be False.

1.2.3.109 TTdbTable.UnlockTable

Removes a previously applied lock on a TurboDB table.

Delphi syntax:

```
procedure UnlockTable(LockType: TTdbLockType);
```

C++ syntax:

```
void __fastcall UnlockTable(TTdbLockType LockType);
```

Description

Call *UnlockTable* to remove a lock previously applied to a database table. LockType specifies the lock to remove.

Removing a write lock enables other applications to read a table. Removing a total lock enables other application to write to a table.

1.2.3.110 TTdbTable.UpdateFullTextIndex

Re-builds a full-text index.

Delphi syntax:

```
procedure UpdateFulltextIndex(const Name: AnsiString); overload;
```

```
procedure UpdateFullTextIndex(const Fields, RelationField,
CounterIndexFileName: String; Limit: Integer); overload;
```

C++ syntax:

```
void __fastcall UpdateFulltextIndex(const AnsiString Name);
```

```
void __fastcall UpdateFullTextIndex(const AnsiString Fields, const
AnsiString RelationField, const AnsiString CounterIndexFileName, int
Limit);
```

Description

Call *UpdateFullTextIndex* to re-build a full-text index for your table. The first version can merely be used with new full-text indexes (table level 4 and above) and is needed only if the index is defect. The second version is for old full-text indexes (table level 3 and below) and is necessary because old full-text indexes are not maintained. You must call this function to add new records to the full-text index and to remove deleted records from it.

Fields is a comma-separated list of the names of the fields you want to be indexed.

RelationField is the name of the relation field, that creates the link to the full-text table.

CounterIndexFileName is optional and points to a text file containing a list of words not to include in the index. The file has one line for each word.

Limit is the number of occurrences of a single keyword that has the keyword excluded from the index.

1.2.3.111 TTdbTable.UpdateIndex

Rebuilds an index of a table.

Delphi syntax:

```
procedure UpdateIndex(const Name: String);
```

C++ syntax:

```
void __fastcall UpdateIndex(const AnsiString Name);
```

Description

Call *UpdateIndex* to rebuild an index of the table. Rebuilding an index can be necessary if the index has faults such that sorting and searching lead to wrong results.

1.2.3.112 TTdbTable.WordFilter

Defines the expression for full-text filtering.

Delphi syntax:

```
property WordFilter: WideString;
```

C++ syntax:

```
__property WideString WordFilter = {read=FWordFilter, write
=SetWordFilter};
```

Description

Set *WordFilter*, if you want to filter the table on one or more keywords. There must be a full-text index for this table and *FullTextTable* must be set to a table component containing the keywords. If both *WordFilter* and *Filter* (or *FilterW*) are set, only records satisfying both conditions can be seen in the table.

Find more on full-text filtering in "Using a Full-text Index at Run-time" and in "Full-text Search-Conditions".

1.2.3.113 TTdbTableFormat

Indicates the file format of a table file.

Unit

TdbTypes

Delphi syntax:

```
type TTdbTableFormat = (tffDBase, tffSDF, tffMyBase, tffTdb);
```

C++ syntax:

```
enum TTdbTableFormat {tffDBase, tffSDF, tffMyBase, tffTdb};
```

Description

Table Format	Description
tffDBase	dBase III+ compatible file
tffSdf	System Data Format. This is a text file where the values are separated by a separator character and enclosed with quote characters.
tffMyBase	XML file in MyBase format. This is the format the TClientDataSet uses to store its data locally. This format is supported for export only.
tffTdb	TurboDB database table file

1.2.3.114 TTdbTableUsage Type

Describes how a database table is currently accessed through applications.

Delphi syntax:

```
TTdbTableUsage = record
  ItemCount: Integer;
  UserCount: Integer;
  ReadCount: Integer;
  UpgradeCount: Integer;
  WriteCount: Integer;
  WaitCount: Integer;
  WaitWriteCount: Integer;
  UpdateCount: Integer;
```

```
  UserInfo: array[0..MaxTableUsers-1] of TTdbUsageUserInfo;
end;
```

C++ syntax:

```
class TTdbTableUsage {
  int ItemCount;
  int UserCount;
  int ReadCount;
  int UpgradeCount;
  int WriteCount;
  int WaitCount;
  int WaitWriteCount;
  int UpdateCount;
  TTdbUsageUserInfo UserInfo[MaxTableUsers];
};
```

Description

ItemCount	Number of applications registered with this table.
UserCount	Number of active sessions working with this table.
ReadCount	Number of sessions currently reading from this table.
UpgradeCount	Number sessions reading from this table and eventually wanting to write to the table.
WriteCount	Number of sessions currently writing to this table.
WaitCount	Number of sessions currently waiting to read from the table.
WaitWriteCount	Number of sessions currently waiting to write to the table.
UpdateCount	Number of modifications applied to the table since the first session accessed it.
UserInfo	Per-session usage information

1.2.3.115 TTdbUsageUserInfo

Describes how a session currently accesses a database table.

Delphi syntax:

```
TTdbUsageUserInfo = record
  ConnectionName: ShortString;
  ConnectionId: LongWord;
  Active: Boolean;
  LockCount: Integer;
  TimeOut: Integer;
  RecordNo: Integer;
end;
```

C++ syntax:

```
class TTdbUsageUserInfo {
  ShortString ConnectionName;
  unsigned int ConnectionId;
  bool Active;
  int LockCount;
  int TimeOut;
  int RecordNo;
};
```

Description

ConnectionName	User-defined name for this session/connection
ConnectionId	System-defined unique id for this session/connection
Active	Session is active
LockCount	Number of read locks

TimeOut	Time out
RecordNo	Physical number of locked record

1.2.3.116 TTdbEncryptionMethod

Indicates the way a table is protected or encrypted.

unit

TdbTypes

Delphi syntax:

```
type TTdbEncryptionMethod = (temDefault, temNone, temProtection,
temClassic, temFastEncrypt, temBlowfish, temRijndael);
```

C++ syntax:

```
enum TTdbEncryptionMethod {temDefault, temNone, temProtection,
temClassic, temFastEncrypt, temBlowfish, temRijndael};
```

Description

The values of this type describe, how a TurboDB database or table is protected against unauthorized access:

Value	Description
temDefault	Allowed only for tables, not for databases. Indicates, that the table uses the general database setting for the encryption method.
temNone	The table is not protected.
temProtection	There is a password for the table, but it is not encrypted.
temClassic	There is a password plus a 32 bit key for encryption.
temFastEncrypt	Fast encryption using a 32 bit key derived from a password.
temBlowfish	Blowfish encryption using a 128 bit key derived from a password.
temRijndael	Rijndael (AES) encryption using a 128 bit key derived from a password.

See also

Data Security

1.2.3.117 TTdbBatchMove

Transfers multiple records between TurboDB tables and other data source e.g. dBase tables.

Unit

TdbBatchMove

Description

TTdbBatchMove imports records from other data sets or from files and exports records into different file types. The file formats TurboDB can work with are:

- Text/SDF
- TurboDB
- dBase
- XML/MyBase (export only)

1.2.3.118 TTdbBatchMove Hierarchy

Hierarchy

TComponent

|

TTdbBatchMove

1.2.3.119 TTdbBatchMove Methods

In TTdbBatchMove

Execute

Derived from TComponent

(check Embarcadero documentation for more information)

1.2.3.120 TTdbBatchMove Properties

In TTdbBatchMove

CharSet
ColumnNames
DataSet
Direction
FileName
FileType
Filter
Mappings
Mode
MovedCount
ProblemCount
Quote
RecalcAutoInc
Separator
TdbDataSet

Derived from TComponent

(check Embarcadero documentation for more information)

1.2.3.121 TTdbBatchMove Events

In TTdbBatchMove

OnProgress

Derived from TComponent

(check Embarcadero documentation for more information)

1.2.3.122 TTdbBatchMove.CharSet

Defines the character set for text files and dBase tables.

Delphi syntax:

```
TTdbCharSet = (tcsAnsi, tcsOem, tcsUtf8);
```

```
property CharSet: TTdbCharSet;
```

C++ Syntax:

```
enum TTdbCharSet {tcsAnsi, tcsOem, tcsUtf8};
```

```
__property Tdbtypes::TTdbCharSet CharSet = {read=FCharSet, write
=FCharSet, nodefault};
```

Description

Set *CharSet* when exporting to a text file or to a dBase file to determine the character set for the exported file. When importing, *CharSet* is important to translate the strings read from the source file.

1.2.3.123 TTdbBatchMove.ColumnNames

Indicates that a text file contains the table column names in the first record.

Delphi syntax:

```
property ColumnNames: Boolean;
```

C++ syntax:

```
__property bool ColumnNames = {read=FColumnNames, write=FColumnNames,
nodefault};
```

Description

This property is only used if the batch move is an export to or an import from a text file. If Direction is set to bmdExport, the property tells the TTdbBatchMove component to write the names of the table fields as the first row in the export file. If Direction is set to bmdImport and ColumnNames is set to true, then the first row of the import file is interpreted as the list of column names.

1.2.3.124 TTdbBatchMove.DataSet

Indicates a data set that is the source for the batch move.

Delphi syntax:

```
property DataSet: TDataSet;
```

C++ syntax:

```
__property Db::TDataSet* DataSet = {read=FDataSet, write=FDataSet};
```

Description

Use *DataSet* when the source for the batch move is a data set rather than a file. If the source is a file, use the FileName property.

1.2.3.125 TTdbBatchMove.Direction

Indicates the direction of the data transfer.

Delphi syntax:

```
TBatchMoveDirection = (bmdImport, bmdExport);
```

```
property Direction: TBatchMoveDirection
```

C++ syntax:

```
enum TBatchMoveDirection {bmdImport, bmdExport};
```

```
__property TBatchMoveDirection Direction = {read=FDirection, write
=FDirection, nodefault};
```

Description

Set this property to switch between import and export of records. Import is a data transfer to *TdbDataSet* and export means transferring data from *TdbDataSet*. Export can only be performed with a file target not with a data set target.

1.2.3.126 TTdbBatchMove.Execute

Performs the batch operation specified by *Direction* and *Mode*.

Delphi syntax:

```
procedure Execute;
```

C++ syntax:

```
void __fastcall Execute(void);
```

Description

After setting properties to indicate what batch operation should be performed and how, call *Execute* to perform the operation. As a minimum, the *FileName*, *FileType, Mode* and *Direction* properties must be set.

1.2.3.127 TTdbBatchMove.FileName

Indicates the file name of the other data source.

Delphi syntax:

```
property FileName: String;
```

C++ syntax:

```
__property AnsiString FileName = {read=FFileName, write=FFileName};
```

Description

Use file name to set the name of the external file for import or export. If the Direction is bmdImport, the file must exist and contain records in the format indicated by the FileType property. If the Direction is *bmdExport*, the file will be created by the *Execute* method. In this case an existing file with this name will be overwritten.

1.2.3.128 TTdbBatchMove.FileType

Indicates the file format of the destination/source data source for the batch move.

Delphi syntax:

```
property FileType: TTdbTableFormat;
```

C++ syntax:

```
__property Tdbtypes::TTdbTableFormat FileType = {read=FTableType, write
=FTableType, nodefault};
```

Description

Set this property to indicate the file type of the data source from which records will be imported or to set the desired file format for the export. Some file types only work in a certain Direction. If the file type does not match the Direction, *Execute* will raise an exception.

1.2.3.129 TTdbBatchMove.Filter

Defines a filter-condition for the records to be imported or exported.

Delphi syntax:

```
property Filter: String;
```

C++ syntax:

```
__property AnsiString Filter = {read=GetFilter, write=SetFilter};
```

Description

Set *Filter* to restrict the set of records to be transferred. Filter is a search-condition as described in "Search-Conditions".

1.2.3.130 TTdbBatchMove.Mappings

Contains a list of column assignments for the import.

Delphi syntax:

```
property Mappings: TStrings;
```

C++ syntax:

```
__property Classes::TStrings* Mappings = {read=FMappings, write
=SetMappings};
```

Description

Set Mappings to specify the correspondence between fields in the Source and fields in the Destination. By default TTdbBatchMove matches fields based on their position in the source and destination tables. That is, the first column in the source is matched with the first column in the destination, and so on. Mappings enables an application to override this default.

To map the column named SourceColName in the source table to the column named DestColName in the destination table, use:

```
DestColName=SourceColName
```

You may also use the column no proceeded by a $ for the mapping:

```
$8=$3
```

You can mix the column identifiers in a mapping as well:

```
$8=SourceColName
```

When adding or appending records, fields in Destination which have no entry in Mappings will be set to NULL. When copying a dataset, fields in Destination which have no entry in Mappings will not appear as columns in the copy of the table.

1.2.3.131 TTdbBatchMove.Mode

Specifies what the *TTdbBatchMove* object does when the Execute method is called.

Delphi syntax:

```
property Mode: TTdbBatchMode;
```

C++ syntax:

```
__property Tdbtypes::TTdbBatchMode Mode = {read=FMode, write=FMode,
nodefault};
```

Description

Use Mode to indicate whether the TTdbBatchMove object should add records, replace records, delete records, or copy the Source. These are the possible values for Mode:

Value	Meaning

tbmAppend	Append the records in the source to the destination table. The destination table must already exist, and the two tables must not have records with duplicate keys. This is the default mode.
tbmUpdate	Replace records in the destination table with matching records from the source table. The destination table must exist and must have an index defined to match records.
tbmAppendUpdate	If a matching record exists in the destination table, replace it. Otherwise, append records to the destination table. The destination table must exist and must have an index defined to match records.
tbmCopy	Create the destination table based on the structure of the source table. If the destination already exists, the operation will delete it, and replace it with the new copy of the source.
tbmDelete	Delete records in the destination table that match records in the source table. The destination table must already exist and must have an index defined.

Note: *TTdbBatchMode* is defined in Unit *TdbTypes.*

1.2.3.132 TTdbBatchMove.MovedCount

Reports the number of records from the Source which were moved to the destination.

Delphi syntax:

```
property MovedCount: LongInt;
```

C++ syntax:

```
__property int MovedCount = {read=FMovedCount, nodefault};
```

Description

Read *MovedCount* to learn the number of records from the Source that were added to the Destination during the *Execute* method. This value does not include records that were excluded from the batch move due to the *Filter* condition.

1.2.3.133 TTdbBatchMove.OnProgress

Occurs after a portion of the batch move process is finished.

Delphi syntax:

```
TTdbProgressEvent = procedure(Sender: TObject; PercentDone: Byte; var
Stop: Boolean) of object;
```

```
property OnProgress: TBatchMoveProgress;
```

C++ syntax:

```
__property Tdbdataset::TTdbProgressEvent OnProgress = {read=FOnProgress,
write=FOnProgress};
```

Description

Write an *OnProgress* event to show the progress of the batch move on screen. *PercentDone* contains the percentage of the records that has been imported/exported. By setting the *Stop* argument to True, you can also halt the batch move.

Note: Do not execute TurboDB database methods in this event handler as this can interfere with the batch move in progress..

1.2.3.134 TTdbBatchMove.ProblemCount

Indicates the number of records which could not be added to Destination without loss of data due to a field type mismatch.

Delphi syntax:

```
property ProblemCount: LongInt;
```

C++ syntax:

```
__property int ProblemCount = {read=FProblemCount, nodefault};
```

Description

Read *ProblemCount* to learn the number of records from the Source that had field values which could not be mapped to destination fields. If the data source is a text file this happens if the string in the input file does not fit the target column, e.g. the string "a" can not be assigned to a date field.

1.2.3.135 TTdbBatchMove.Quote

Defines the character used to enclose strings in a text file.

Delphi syntax:

```
property Quote: Char;
```

C++ syntax:

```
__property char Quote = {read=FQuote, write=FQuote, nodefault};
```

Description

Set *Quote* to a value different from #0 to enclose strings in the exported file. When importing, *Quote* is important to determine the correct values for strings from the import file. *Quote* only matters when *FileType* is set to *tffSDF*.

1.2.3.136 TTdbBatchMove.RecalcAutoInc

Specifies whether the AutoInc values should be calculated anew during import.

Delphi Syntax:

```
property RecalcAutoInc: Boolean;
```

C++ Syntax:

```
__property bool RecalcAutoInc = {read=FRecalcAutoInc, write=
FRecalcAutoInc, default=0}
```

Description

Set this property to True, if you want to import records containing AutoInc values and if those records have values already present in the destination table. If the property is set to true, TurboDB will calculate new unique values for the imported AutoInc fields. If the property is set to false, TurboDB will try to keep the AutoInc values from the source table. In this case it may happen, that your table contains duplicate AutoInc values. Use *SetNextAutoIncValue* to define, at which point the auto-increment values shall be continued after the import.

Note: If your application relies on the imported AutoInc values for table relationships, you will loose the links between the tables, if you set *RecalcAutoInc* to true.

1.2.3.137 TTdbBatchMove.Separator

Defines the character used to separate the field values in a text file.

Delphi syntax:

```
property Separator: Char;
```

C++ syntax:

```
__property char Separator = {read=FSeparator, write=FSeparator,
nodefault};
```

Description

Set *Separator* to properly analyze the field values in a text file to import. When exporting, *Separator* determines the separator character to create for the output file. *Separator* is needed only for text files i.e. if *FileType* is set to *tffSDF*.

1.2.3.138 TTdbBatchMove.TdbDataSet

Indicates the TurboDB table component, that is the source or destination for export or import.

Delphi syntax:

```
property TdbDataSet: TTdbDataSet;
```

C++ syntax:

```
__property Tdbdataset::TTdbDataSet* TdbDataSet = {read=FTdbDataSet,
write=FTdbDataSet};
```

Description

Set *TdbDataSet* to the data set component into which you want transfer records or from which you want to export records. If Direction is *bmdImport*, records will be transferred from the external data source FileName or DataSet in to the table indicated by *TdbDataSet*. If *Direction* is *bmdExport*, the records will be written in the file from the data set component.

1.2.3.139 TTdbDatabase

TTdbDatabase provides discrete control over a connection to a single database in a TurboDB-based database application.

Unit

TdbDataSet

Description

Use TTdbDatabase when a TurboDB-based database application requires any of the following control over a database connection:

- Common database directory for two or more TurboDB data sets
- Providing a password-dialog to open protected tables
- Reestablishing formerly used connections

Note

Explicit declaration of a *TTdbDatabase* component for each database connection in an application is optional if the application does not need to explicitly control that connection. If a *TTdbDatabase* component is not explicitly declared and instantiated for a database connection, a temporary database component with a default set of properties is created for it at runtime.

1.2.3.140 TTdbDatabase Hierarchy

Hierarchy

TObject

|

TPersistent

|

TComponent

|

TCustomConnection

|

TTdbDatabase

1.2.3.141 TTdbDatabase Methods

In TTdbDatabase

Backup
CloseCachedTables
CloseDataSets
Commit
Compress
RefreshDataSets
Rollback
StartTransaction

Derived from TCustomConnection

(check Embarcadero documentation for more information)

1.2.3.142 TTdbDatabase Properties

In TTdbDatabase

BlobBlockSize
CacheSize
ConnectionId
ConnectionName
DatabaseName
Exclusive
FlushMode
IndexPageSize
Location
PrivateDir

Derived from TCustomConnection

(check Embarcadero documentation for more information)

1.2.3.143 TTdbDatabase Events

In TTdbDatabase

OnPassword

Derived from TCustomConnection

(check Embarcadero documentation for more information)

1.2.3.144 TTdbDatabase.BlobBlockSize

Specifies the block size of blob objects for the next table to be created.

Delphi syntax:

```
property BlobBlockSize: Integer;
```

C++ syntax:

```
__property int BlobBlockSize = {read=FBlobBlockSize, write
=FBlobBlockSize, nodefault};
```

Description

The block size is the allocation unit for blob objects. TurboDB sets a reasonable default value for this parameter but you may optimize it in special cases. For example, if the table will store mainly very large images, you might want to set the block block size to 16 K or 64 K or even more. This size will be used in all subsequent calls to *TTdbTable.CreateTable* and *TTdbTable.AlterTable*.

You may want to experiment with different values to find out the block size for best performance.

1.2.3.145 TTdbDatabase.Backup

Backs-up the database to the indicated target location.

Delphi syntax:

```
procedure Backup(TargetLocation: string; DatabaseType:
TTdbDatabaseType);
```

C++ syntax:

```
void __fastcall Backup(const AnsiString TargetLocation, TTdbDatabaseType
DatabaseType);
```

Description

The database type of the backup database is independent of the original database. It only depends on the *DatabaseType* argument. The backup location - target directory or target file depending on the database type - must not yet exist. Backup can be called during normal database operation. It tries to obtain a consistent state of the database and copies it to the target location. If it does not succeed within the *LockingTimeout*, an exception is thrown.

1.2.3.146 TTdbDatabase.AutoCreateIndexes

Specifies, if a query automatically creates temporary indexes during execution.

Delphi syntax:

```
property AutoCreateIndexes: Boolean;
```

C++ syntax:

```
__property bool AutoCreateIndexes = {read=FExclusive, write
=SetExclusive, nodefault};
```

Description

If AutoCreateIndexes is set to *True* (default), executing a query can create temporary indexes to be needed for faster execution. These indexes will be automatically removed when the last session stops accessing the database.

1.2.3.147 TTdbDatabase.CacheSize

Specifies the amount of memory to use for database caching.

Delphi syntax:

```
property CacheSize: Integer;
```

C++ syntax:

```
__property int CacheSize;
```

Description

Set this value to a lower value to leave more memory for other applications. Set it to a higher value to improve the performance of database operations.

1.2.3.148 TTdbDatabase.CloseCachedTables

Physically closes all data files which are currently not in use.

Delphi syntax:

```
procedure CloseCachedTables;
```

C++ syntax:

```
void __fastcall CloseCachedTables(void);
```

Description

Closing a dataset does not automatically close the corresponding data files. This speeds up access times if the data will be needed again.

Executing *CloseCachedTables* physically closes all data files associated with this database component which are currently unused.

This method can be needed if you try to execute file operations on data files eg. for backup/restore.

1.2.3.149 TTdbDatabase.CloseDataSets

Closes all datasets.

Delphi syntax:

```
procedure CloseDataSets;
```

C++ syntax:

```
void __fastcall CloseDataSets(void);
```

Description

CloseDataSets closes all datasets associated with this database component.

1.2.3.150 TTdbDatabase.Commit

Finishes a transaction and commits all modifications to the database.

Delphi syntax:

```
procedure Commit;
```

C++ syntax:

```
void __fastcall Commit(void);
```

Description

Use *Commit* to persist all modifications of the current transaction. The procedure throws an exception, if no transaction has been started using *StartTransaction*.

1.2.3.151 TTdbDatabase.Compress

Reduces the file size of a single-file database to the minimum.

Delphi syntax:

```
procedure Compress;
```

C++ Syntax:

```
void __fastcall Compress(void);
```

Description

A single-file database reuses the space gained from the deletion of records for new records, but it does not free this space. Therefore the file size of a single-file database is not reduced, even if all records of all tables are deleted. While this behavior is the best for optimal performance, it is sometimes not desired, e.g. for deployment of the database. Calling *Compress* frees this unused disk space.

You must make sure that there is absolutely no other access to the database, when you call *Compress*. The safes way to reach this is to close the database and reopen it in exclusive mode.

The method has no effect on a directory database.

1.2.3.152 TTdbDatabase.ConnectionId

Indicates the id that identifies this database connection.

Delphi syntax:

```
property ConnectionId: Integer;
```

C++ Syntax:

```
__property unsigned ConnectionId = {read=FConnectionId, nodefault};
```

Description

Read this property to determine the id of the database connection you are using. Every time the TTdbDatabase object connects to the TurboDB engine it is assigned a unique connection id. This connection id is used in lock files to identify the locking database connection.

1.2.3.153 TTdbDatabase.ConnectionName

Specifies a readable name for the database connection.

Delphi syntax:

```
property ConnectionName: String;
```

C++ syntax:

```
__property String ConnectionName = {read=FConnectionName, write
=FConnectionName};
```

Description

Set this property to define a readable name for the TurboDB connection this database object represents. This name is useful, if you want to show a list of current users of a table to humans.

1.2.3.154 TTdbDatabase.DatabaseName

Assigns a name to this database connection.

Delphi syntax:

```
property DatabaseName: String;
```

C++ syntax:

```
__property AnsiString DatabaseName = {read=FDatabaseName, write
=SetDatabaseName};
```

Description

Use *DatabaseName* to specify the name for the database. You may choose any name you want, but it must be unique for databases within the current application. The *DatabaseName* must match the *DatabaseName* of the *TTdbDataSet* components that use this database.

Note: Attempting to set *DatabaseName* when the *Connected* property is True raises an exception.

1.2.3.155 TTdbDatabase.Exclusive

Enables an application to gain sole access to a database.

Delphi syntax:

```
property Exclusive: Boolean;
```

C++ syntax:

```
__property bool Exclusive = {read=FExclusive, write=SetExclusive,
nodefault};
```

Description

Opens the database exclusively, i.e. other applications cannot access its tables.

1.2.3.156 TTdbDatabase.FlushMode

Specifies the default flush mode for the tables of this database.

Delphi syntax:

```
property FlushMode: TTdbFlushMode;
```

C++ syntax:

```
__property TTdbFlushMode FlushMode = {read=FFlushMode, write=FFlushMode,
nodefault};
```

Description

Set this property to define the flush mode for all tables of this database. The value *tfmDefault* on database level is identical to *tfmFast*.

1.2.3.157 TTdbDatabase.IndexPageSize

Specifies the page size of the next index to be created.

Delphi syntax:

```
property IndexPageSize: Integer;
```

C++ syntax:

```
__property int IndexPageSize = {read=FIndexPageSize, write
=FIndexPageSize, nodefault};
```

Description

Usually TurboDB calculates the size of its index pages internally. Using this property you may optimize the index in special cases. The property specifies the size of a B-tree page in bytes. This size will be used for all subsequent index creations using *TTdbTable.AddIndex*.

The smaller the index pages are, the higher the index B-tree will become. Normally, you should choose the size of the index page such that between 10 and 300 table rows can be indexed per page. If in doubt, experiment with different sizes and find out, which one is the fastest.

1.2.3.158 TTdbDatabase.Location

Specifies the location of the database.

Delphi syntax:

```
property Location: String;
```

C++ syntax:

```
__property TTdbFlushMode FlushMode = {read=FFlushMode, write=FFlushMode,
nodefault};
```

Description

Set *Location* to define the TurboDB database to use. Location is either a database directory where database tables are searched by default (this is called a directory database) or the name of a TurboDB single-file database file (this is called a single-file database). Single-file databases have the extension *tdbd*.

Using a relative filename (preceding dot) is not allowed and will cause an exception.

Note

At design-time, double-click on the *TTdbDatabase* component to invoke the component editor and browse for a database location.

1.2.3.159 TTdbDatabase.LockingTimeOut

Defines the time to wait for a locked record.

Delphi syntax:

```
property LockingTimeout: Integer;
```

C++ syntax:

```
__property int LockingTimeout = {read=FLockingTimeout, write
=FLockingTimeout, nodefault};
```

Description

When a session wants to edit a record which is already being edited by another session, TurboDB waits for time defined by this property before the second session receives an exception. The waiting time is given in milliseconds.

1.2.3.160 TTdbDatabase.OnPassword

Occurs when an application attempts to open a protected TurboDB table for the first time.

Delphi syntax:

```
TTdbPasswordEvent = procedure(Sender: TObject; const TableName: string;
var Key: WideString; var Retry: Boolean) of object;
```

```
property OnPassword: TTdbPasswordEvent;
```

C++ syntax:

```
typedef void __fastcall (__closure *TTdbPasswordEvent)(System::TObject*
Sender, const AnsiString TableName, WideString &Key, bool &Retry);
```

```
__property TTdbPasswordEvent OnPassword = {read=FOnPassword, write
=FOnPassword};
```

Description

Write an OnPassword event handler to take specific action when an application attempts to open a password-protected or encrypted TurboDB table for the first time. To gain access to the TurboDB table, the event handler must set the parameters to *Key* . Use Retry to conditionally finalize opening the table. If Retry is set to True, opening the table is tried with the new password provided in the arguments. If set to False, the attempt to open the table is abandoned.

Note

If an *OnPassword* event handler does not exist, but TurboDB reports insufficient access rights, an exception is raised.

Example

The following example shows a typical event handler and the source code for opening the table.

```
procedure TForm1.Password(Sender: TObject; const TableName: string; var
Key: WideString; var Retry: Boolean);
begin
     Key := InputBox('Enter password', 'Password:', '');
     Retry := (Key > '');
end;
```

```
procedure TForm1.OpenTableBtnClick(Sender: TObject);
begin
     Database.OnPassword := Password;
     try
          Table1.Open;
     except
          if not Table1.Active then begin
               ShowMessage('Could not open table');
               Application.Terminate;
    end;
  end;
end;
```

1.2.3.161 TTdbDatabase.PrivateDir

Specifies the directory in which to store temporary table processing files generated by TurboDB for database components associated with this database.

Delphi syntax:

```
property PrivateDir: String;
```

C++ syntax:

```
__property AnsiString PrivateDir = {read=FPrivateDir, write
=FPrivateDir};
```

Description

Use PrivateDir to set the directory in which to store temporary table processing files for all database connections, such as those generated by TurboDB to handle local SQL statements. Ordinarily this value is only set at runtime, so that a user's local hard disk is used to store temporary files. Local storage of these files improves performance. If no value is specified for PrivateDir, TurboDB automatically stores those files in the Windows temporary directory. If you set this property be sure not to share the private directory between different applications or users.

Note

For applications that run directly from a networked file server, the application should set *PrivateDir* to a user's local drive to improve performance and to prevent temporary files from being created on the server where they might conflict with temporary files created by other instances of the application.

1.2.3.162 TTdbDatabase.RefreshDataSets

Updates data access components that have been changed by other users.

Delphi syntax:

```
procedure RefreshDataSets;
```

C++ syntax:

```
void __fastcall RefreshDataSets(void);
```

Description

Use *RefreshDataSets* to refresh all datasets associated with this database component that are changed by other databases/applications in a multi-user environment. *RefreshDataSets* checks which datasets have been modified since the last update and updates the datasets as necessary.

The typical use of *RefreshDataSets* is in an OnTimer handler. Every second or so you can refresh the datasets of the database to reflect changes made in the work group environment.

1.2.3.163 TTdbDatabase.Rollback

Terminates a transaction and cancels all modifications.

Delphi syntax:

```
procedure Rollback;
```

C++ syntax:

```
void __fastcall Rollback(void);
```

Description

Use *Rollback* to take back all modifications in the current transaction. The procedure raises an exception if no transaction has been started using *StartTransaction*.

1.2.3.164 TTdbDatabase.StartTransaction

Begins a transaction.

Delphi syntax:

```
procedure StartTransaction;
```

C++ syntax:

```
void __fastcall StartTransaction(void);
```

Description

Use *StartTransaction* to begin a new transaction with isolation level read committed. Since TurboDB can only have one active transaction per database session at given time, the procedure will throw an exception, if there is already a transaction running. Call *Commit* or *Rollback* to end a transaction.

1.2.3.165 TTdbEnumValueSet

TTdbEnumValueSet provides access to the capabilities of enumeration fields.

Unit

TdbEnumValueSet

Description

Use *TTdbEnumValueSet* to

- provide a combo box offering the enumeration values for input in forms and grids
- define alias names for the enumeration values (e.g. to translate your application)

TTdbEnumValueSet is a TDataSet descendant for use with lookup fields. This enables you to integrate enumeration field support with any data-aware control supporting lookup.

1.2.3.166 TTdbEnumValueSet Hierarchy

Hierarchy

TObject
|
TPersistent
|
TComponent
|
TDataSet
|
TTdbEnumValueSet

1.2.3.167 TTdbEnumValueSet Properties

In TTdbEnumValueSet

DataSource

EnumField

Values

Derived from TDataSet

(check Embarcadero documentation for more information)

1.2.3.168 TTdbEnumValueSet.DataSource

Identifies the data source of the enumeration field.

Delphi syntax:

```
property DataSource: TDataSource;
```

C++ syntax:

```
__property Db::TDataSource* MasterSource = {read=FMasterSource, write
=SetDataSource};
```

Description

Set DataSource to link the EnumValueSet to the TdbDataSet which contains the enumeration field.

1.2.3.169 TTdbEnumValueSet.EnumField

Identifies the enumeration field in DataSource whose values are shown.

Delphi syntax:

```
property EnumField: String;
```

C++ syntax:

```
__property AnsiString EnumField = {read=GetDataField, write
=SetDataField};
```

Description

Set *EnumField* to identify the enumeration field in DataSource.

1.2.3.170 TTdbEnumValueSet.Values

Holds the enumeration values and their aliases.

Delphi syntax:

```
property Values: TStrings;
```

C++ syntax:

```
__property Classes::TStrings* Values = {read=FItems, write=SetValues};
```

Description

You can use Values to define alias names for the enumeration values. The alias names are the ones shown in combo boxes and can be translated or changed using the Values property.

1.2.3.171 TTdbQuery

TTdbQuery issues SQL statements for the TurboDB Engine and accesses the result set.

Unit

TdbQuery

Description

Use *TTdbQuery* to create a result set from one or more database tables using a SQL select statement or to edit a database table via a INSERT, DELETE or UPDATE statement. Refer to the "TurboSQL Guide" for the applicable SQL syntax.

Remark

The *TTdbQuery* component is included in the professional edition of TurboDB only.

1.2.3.172 TTdbQuery Hierarchy

Hierarchy

TObject

|

TPersistent

|

TComponent

|

TDataSet

|

TTdbDataSet

|

TTdbQuery

1.2.3.173 TTdbQuery Events

Derived from TTdbDataSet

OnProgress

OnResolveLink

Derived from TDataSet

(check Embarcadero documentation for more information)

1.2.3.174 TTdbQuery Methods

In TTdbQuery

ExecSQL

Prepare

UnPrepare

Derived from TTdbDataSet

GetEnumValue

Locate

Lookup

Replace

Derived from TDataSet

(check Embarcadero documentation for more information)

1.2.3.175 TTdbQuery Properties

In TTdbQuery

Params

RequestStable

SQL

SQLW

UniDirectional

UnPrepare

Derived from TTdbDataSet

FieldDefsTdb

Filter

Filtered

Version

Derived from TDataSet

(check Embarcadero documentation for more information)

1.2.3.176 TTdbQuery.ExecSQL

Executes the SQL statement for the query.

Delphi syntax:

```
procedure ExecSQL;
```

C++ syntax:

```
void __fastcall ExecSQL(void);
```

Description

Call *ExecSQL* to execute the SQL statement currently assigned to the SQL property. Use *ExecSQL* to execute queries that do not return a cursor to data (such as INSERT, UPDATE, DELETE, and CREATE TABLE).

Note

For SELECT statements, call *Open* instead of *ExecSQL*.

ExecSQL prepares the statement in SQL property for execution if it has not already been prepared. To speed performance, an application should ordinarily call *Prepare* before calling *ExecSQL* for the first time.

1.2.3.177 TTdbQuery.Params

Contains the parameters for a query's SQL statement.

Delphi syntax:

```
property Params[Index: Word]: TParams;
```

C++ syntax:

```
__property Db::TParams* Params = {read=FParams, write=SetParamsList,
stored=false};
```

Description

Access Params at runtime to view and set parameter names, values, and data types dynamically (at design time use the collection editor for the *Params* property to set parameter information). *Params* is a zero-based array of *TParams* parameter records. Index specifies the array element to access.

Note

An easier way to set and retrieve parameter values when the name of each parameter is known is to call *ParamByName*. *ParamByName* cannot, however, be used to change a parameter's data type or name.

Parameters used in SELECT statements cannot be NULL, but they can be NULL for UPDATE and INSERT statements.

1.2.3.178 TTdbQuery.Prepare

Sends a query to the TurboDB Engine for optimization prior to execution.

Delphi syntax:

```
procedure Prepare;
```

C++ syntax:

```
void __fastcall Prepare(void);
```

Description

Call *Prepare* to have TurboDB allocate resources for the query and to perform additional optimizations. Calling *Prepare* before executing a query improves application performance.

Delphi automatically prepares a query if it is executed without first being prepared. After execution, Delphi unprepares the query. When a query will be executed a number of times, an application

should always explicitly prepare the query to avoid multiple and unnecessary prepares and unprepares.

Preparing a query consumes some database resources, so it is good practice for an application to unprepare a query once it is done using it. The *UnPrepare* method unprepares a query.

Note

When you change the text of a query at runtime, the query is automatically closed and unprepared.

1.2.3.179 TTdbQuery.RequestStable

Requests a query result set, that does not change when rows are modified.

Delphi syntax:

```
property RequestStable: Boolean;
```

C++ syntax:

```
__property bool RequestStable = {read=FRequestStable, write
=FRequestStable, nodefault};
```

Description

If possible, executing the query will create a stable result set, which is not reordered or shrunk when rows are edited. For example, when a value is modified which contributes to the sort order, the row is not moved to another position even if the sort order is now violated.

1.2.3.180 TTdbQuery.SQL

Contains the text of the SQL statement to execute for the query.

Delphi syntax:

```
property SQL: TStrings;
```

C++ syntax:

```
__property Classes::TStrings* SQL = {read=FSQL, write=SetQuery};
```

Description

SQL provides access to the SQL statement compatible with the Object Inspector, with Embarcadero TQuery and with previous versions of TurboDB. Use *SQLW* if your SQL statement contains true Unicode characters or if you don't want to work with *TStrings*.

1.2.3.181 TTdbQuery.SQLW

Contains the Unicode text of the SQL statement to execute for the query.

Delphi syntax:

```
property SQLW: WideString;
```

C++ syntax:

```
__property WideString SQLW = {read=FSQLW, write=SetQueryW};
```

Description

Use *SQLW* to provide the SQL statement that a query component executes when its *ExecSQL* or *Open* method is called. At design time the *SQLW* property can be edited by invoking the SQLBuilder in the Object Inspector.

The *SQLW* property may contain one or more SQL commands separated by semicolon ";". If there are multiple result sets in a chained SQL statement, only the last one will be shown in the query component.

The *SQLW* property can be used to access TurboDB tables using TurboSQL. The allowable syntax is a subset of SQL-92.

The SQL statement in the *SQLW* property may contain replaceable parameters, following standard SQL-92 syntax conventions. Parameters are created and stored in the Params property.

1.2.3.182 TTdbQuery.UniDirectional

Determines whether or not TurboDB Engine bidirectional cursors are enabled for a query's result set.

Delphi syntax:

```
property UniDirectional: Boolean;
```

C++ syntax:

```
__property bool UniDirectional = {read=FUniDirectional, write
=FUniDirectional, nodefault};
```

Description

Since TurboDB cursors are always bidirectional the value of this property doesn't matter. It is provided for BDE-compatibility only.

1.2.3.183 TTdbQuery.UnPrepare

Frees the resources allocated for a previously prepared query.

Delphi syntax:

```
procedure UnPrepare;
```

C++ syntax:

```
void __fastcall UnPrepare(void);
```

Description

Call *UnPrepare* to free the resources allocated for a previously prepared query.

Preparing a query consumes some database resources, so it is good practice for an application to unprepare a query once it is done using it. The *UnPrepare* method unprepares a query.

Note

When you change the text of a query at runtime, the query is automatically closed and unprepared.

1.2.3.184 TTdbFieldDef

TTdbFieldDef is a field definition that corresponds to a physical field of a record in a TurboDB table underlying a dataset.

Unit

TdbDataSet

Description

A *TTdbFieldDef* object contains the definition of one field in a table. The definition for a field includes such attributes as the field's name, data type, and size. *TTdbFieldDef* objects are typically used in collections of such objects, such as the *FieldDefsTdb* property of the *TTdbDataSet* component.

When using an existing table, a field definition is automatically created for each field in the dataset. Inspect the properties of *TTdbFieldDef* to retrieve information about specific fields in the dataset.

When creating new tables, such as with the *TTdbTable.CreateTable* method, *TTdbFieldDef* objects are used to supply the definitions for the new fields that will comprise the new table.

A field definition has a corresponding *TField* object, but not all *TField* objects have a corresponding field definition object. For example, calculated fields do not have field definition objects.

There are two primary reasons for working with *TTdbFieldDef* objects:

- To obtain information about field types in a table without opening the table.
- To specify field definitions for a new table or for altering a table.

1.2.3.185 TTdbFieldDef Hierarchy

Hierarchy

TObject

|

TPersistent

|

TCollectionItem

|

TTdbFieldDef

1.2.3.186 TTdbFieldDef Properties

In TTdbFieldDefs

Expression

FieldNo

InitialFieldNo

InternalCalcField

DataTypeTdb

Specification

Derived from TDefCollection

(check Embarcadero documentation for more information)

1.2.3.187 TTdbFieldDef Methods

In TTdbFieldDef

Assign

Derived from TCollectionItem

(check Embarcadero documentation for more information)

1.2.3.188 TTdbFieldDef.Assign

Copies the contents of another field definition.

Delphi syntax:

```
procedure Assign(Source: TPersistent); override;
```

C++ syntax:

```
virtual void __fastcall Assign(Classes::TPersistent* Source);
```

Description

Use *Assign* to assign the values of another *TTdbFieldDef* object or a *TFieldDef* object to the *TTdbFieldDef* object executing the method.

1.2.3.189 TTdbFieldDef.DataTypeTdb

Holds the native data type of the underlying field.

Delphi syntax:

```
type TTdbDataType = (dtUnknown, dtString, dtInteger, dtSmallInt,
dtFloat, dtByte, dtBoolean, dtDate, dtMemo, dtEnum, dtAutoInc, dtLink,
dtBlob, dtTime, dtRelation, dtDateTime, dtWideString, dtWideMemo,
dtLargeInt, dtGuid); // dtGUID only available for Delphi 5 or higher
```

```
property DataTypeTdb: TTdbDataType;
```

C++ syntax:

```
enum TTdbDataType {dtUnknown, dtString, dtInteger, dtSmallInt, dtFloat,
dtByte, dtBoolean, dtDate, dtMemo, dtEnum, dtAutoInc, dtLink, dtBlob,
dtTime, dtRelation, dtDateTime, dtWideString, dtWideMemo, dtLargeInt,
dtGuid}; // dtGUID only available for C++ Builder 5 or higher
```

```
__property TTdbDataType DataTypeTdb = {read=GetDataTypeTdb, write
=SetDataTypeTdb, nodefault};
```

Check *DataTypeTdb* to find out the true native data type of the field. Set *DataTypeTdb* to create a table column using one of the non-standard TurboDB field types.

1.2.3.190 TTdbFieldDef.CalcExpression

Holds the expression if the field is calculated.

Delphi syntax:

```
property CalcExpression: String;
```

C++ syntax:

```
__property AnsiString CalcExpression = {read=FCalcExpression, write
=FCalcExpression};
```

Description

Use *CalcExpression* to specify a calculated field in a TurboDB table. The field *InternalCalcField* decides, whether the calculation is just an initial calculation (i.e. a default value) or a permanent calculation. If *InternalCalcField* is True, the value of the field is calculated each time the record changes and is then stored into the table. If *InternalCalcField* is False, the expression is used to initialize the column value, which can be modified afterwards.

Example

The following adds a field definition to calculate the product price including 16% taxes:

```
with TdbTable1.FieldDefsTdb.Add('Price', dtFloat) do begin
     InternalCalcField := True;
     CalcExpression := 'NetPrice * 1.16';
end;
```

See also

TTdbFieldDef.InternalCalcField property

1.2.3.191 TTdbFieldDef.FieldNo

Identifies the physical field number used to reference the field.

Delphi syntax:

```
property FieldNo: Integer;
```

C++ syntax:

```
__property int FieldNo = {read=GetFieldNo, write=SetFieldNo, nodefault};
```

Description

Use *FieldNo* to find out where the physical field the field definition references is in the set of fields in the table. For example, if the value of *FieldNo* is 2, then the field is the second field in the underlying database table.

When adding field definitions to a dataset, set *FieldNo* to specify the location at which to create the field.

The difference between *FieldNo* and InitialFieldNo is that *InitialFieldNo* represents the current status of the database table, whereas *FieldNo* is the field position, which will be valid after the next call to *CreateTable* or *AlterTable*.

1.2.3.192 TTdbFieldDef.InitialFieldNo

Holds the position of the physical field in the table.

Delphi syntax:

```
property InitialFieldNo: Integer;
```

C++ syntax:

```
__property short InitialFieldNo = {read=FInitialFieldNo, write
=FInitialFieldNo, nodefault};
```

Description

InitialFieldNo is a read-only property and is used internally to track column movements within the field definitions.

1.2.3.193 TTdbFieldDef.InternalCalcField

Specifies whether the expression is used for the default value or a calculated column.

Delphi syntax:

```
property InternalCalcField: Boolean;
```

C++ syntax:

```
__property bool InternalCalcField = {read=FInternalCalcField, write
=FInternalCalcField, nodefault};
```

Description

Set *InternalCalcField* to True if you want the expression in *CalcExpression* be calculated each time a record is modified. As a result, the column will always contain a value which the result of the *CalcExpression*. Set *InternalCalcField* to *False* if the result of the expression is a default value for new records and shall not be evaluated later on.

1.2.3.194 TTdbFieldDef.Specification

Holds additional information for the field definition according to the field type.

Delphi syntax:

```
property Specification: string;
```

C++ syntax:

```
__property AnsiString Specification = {read=FSpec, write=FSpec};
```

Description

Specification has different meanings depending on the field type of the field definition:

- For textual fields (string, wide string, memo, wide memo), *Specification* holds the name of the collation. If the property is an empty string, the collation *TurboDB* is used.
- For enumeration fields *Specification* holds the enumeration values in a comma-separated format.
- For link and relation fields *Specification* defines the name of the table to link to.
- Field definitions of auto-increment fields save the indication in this property. This is a field name or a comma-separated list of fields of the table associated with the *TTdbFieldDef* which define the values to display as a readable reference to the linked record in link and relation fields. (See "The Concept of Automatic Linking")

1.2.3.195 TTdbFieldDefs

TTdbFieldDefs holds the field definition (*TTdbFieldDef*) objects that represent the physical fields underlying a dataset.

Unit

TdbDataSet

Description

TTdbFieldDefs is used by a TurboDB dataset to manage the field definitions it uses to create field objects that correspond to fields in the database table. *TTdbTable* also uses *TTdbFieldDefs* when creating a new database table.

Use the properties and methods of *TTdbFieldDefs* to:

- Access a specific field definition.
- Add or delete field definitions from the list (new tables).
- Find out how many fields are defined.
- Copy a set of field definitions to another table.

1.2.3.196 TTdbFieldDefs Hierarchy

Hierarchy

TObject

|

TPersistent

|

TCollection

|

TOwnedCollection

|

TDefCollection

|

TTdbFieldDefs

1.2.3.197 TTdbFieldDefs Methods

In TTdbFieldDefs

Add

Assign

Find

Derived from TDefCollection

(check Embarcadero documentation for more information)

1.2.3.198 TTdbFieldDefs Properties

In TTdbFieldDefs

Items

Derived from TDefCollection

(check Embarcadero documentation for more information)

1.2.3.199 TTdbFieldDefs.Add

Creates a new field definition object and adds it to the Items property of this *TTdbFieldDefs* object.

Delphi syntax:

```
function Add(const Name: String; DataTypeTdb: TTdbDataType; Size:
Integer = 0; Required: Boolean = False; const Specification: string = ''
): TTdbFieldDef;
```

C++ syntax:

```
TTdbFieldDef* __fastcall Add(const AnsiString Name, TTdbDataType
DataTypeTdb, int Size = 0, Boolean Required = false, const AnsiString
Specification);
```

Description

Use *Add* to add a new *TdbFieldDef* to the list of field definitions. Add returns a reference to the new *TTdbFieldDef* object within the list.

Add uses the values passed in the Name, *DataType*, *Size*, *Required* and *Specification* parameters and assigns them to the respective properties of the new field definition object.

If a field definition with same name already exists, an *EDatabaseError* exception is raised.

Example

The following code clears the *FieldDefsTdb* list and adds three fields for altering the table:

```
// Clear the field definition list
TdbTable1.FieldDefsTdb.Clear;
// Add a string field called name 40 characters long
TdbTable1.FieldDefsTdb.Add('Name', dtString, 40);
{ Add an enumeration field that has the enumeration values "Sales",
"Marketing", "Development" }
with TdbTable1.FieldDefsTdb.Add('Department', dtEnum) do Specification
:= 'Sales,Marketing,Development';
{ Add an auto-incrementing field used as a record id. References to
records of this table via this auto-incrementing field are displayed as
the Name of the person. }
with TdbTable1.FieldDefsTdb.Add('RecordId', dtAutoInc) do Specification
:= 'Name';
// The table is restructured to reflect the new field definitions
TdbTable1.AlterTable;
```

See also

TTdbFieldDef

1.2.3.200 TTdbFieldDefs.Assign

Copies the contents of another field definition list to the object where the method is executed.

Delphi syntax:

```
procedure Assign(Source: TPersistent); override;
```

C++ syntax:

```
virtual void __fastcall Assign(Classes::TPersistent* Source);
```

Description

Use Assign to copy the contents of one TTdbFieldDefs instance to another. The Assign method deletes all items from the destination collection (the object where it is executed), then adds a copy of each item in the Source collection's Items array. You can assign either a TTdbFieldDefs object or a TFieldDefs object to an instance of TTdbFieldDefs. Assign is mainly employed to create a table that has the same fields as another already existing table.

1.2.3.201 TTdbFieldDefs.Find

Locates a definition object in the Items array from its name.

Delphi syntax:

```
function Find(const AName: string): TTdbFieldDef;
```

C++ syntax:

```
TTdbFieldDef* __fastcall Find(const AnsiString Name);
```

Description

Call *Find* to obtain information about a particular field definition object. Specify the name of the field definition object as the value of the Name parameter.

1.2.3.202 TTdbFieldDefs.Items

Lists the field definitions that describe each physical field in the dataset.

Delphi syntax:

```
property Items[Index: Integer]: TTdbFieldDef; default;
```

C++ syntax:

```
__property TTdbFieldDef* Items[int Index] = {read=GetFieldDef, write
=SetFieldDef};
```

Description

Use *Items* to access a particular field definition. Specify the field definition to access with the *Index* parameter. Index is an integer identifying the field definition's position in the list of field definitions, in the range 0 to Count - 1. This property is very close to standard VCL *TFieldDefs*.Items but holds *TTdbFieldDef* objects instead of *TFieldDef* objects.

1.2.3.203 TTdbFlushMode

Indicates the kind of write buffering.

Delphi syntax:

```
TTdbFlushMode = (tfmDefault, tfmSecure, tfmFast);
```

C++ syntax:

```
enum TTdbFlushMode {tfmDefault, tfmSecure, tfmFast};
```

Description

These values are applied to TTdbDatabase.FlushMode and TTdbTable.FlushMode.

1.2.3.204 TTdbLockType

Indicates the kind of lock to apply.

Delphi syntax:

```
TTdbLockType = (tltWriteLock, tltTotalLock);
```

C++ syntax:

```
enum TTdbLockType {tltWriteLock, tltTotalLock};
```

Description

tltWriteLock A write lock prevents other applications from writing to the table

Use this type of lock if you want to ensure that no other application can write changes to the table during your read operation.

tltTotalLock A total lock prevents other applications from accessing the table

Use this type of lock if you want to ensure that no other application can read from the table during your write operation.

1.2.3.205 TTdbBlobProvider Class

TTdbBlobProvider support displaying images of different formats read from a database blob column.

Unit

TdbExtComps

Description

When you connect a blob provider to a blob column of a data source, the blob provider tries to detect the file format of the current blob and loads it. It is extensible, so you can add further graphics formats, for which a *TGraphic* descendant exists. It can also be used to write images into database blobs and even link images to database blobs.

See also

Images demo program

1.2.3.206 TTdbBlobProvider Hierarchy

Hierarchy

TObject

|

TPersistent

|

TComponent

|

TTdbBlobProvider

1.2.3.207 TTdbBlobProvider Events

Events in TTdbBlobProvider

OnReadGraphic

OnUnknownFormat

Derived from TComponent

(check Borland/CodeGear/Embarcadero documentation for more information)

1.2.3.208 TTdbBlobProvider Methods

In *TTdbBlobProvider*

Create

Destroy

RegisterBlobFormat

CreateTextualBitmap

LoadBlob

SetBlobData

SetBlobLinkedFile

Derived from *TComponent*

(check Borland/CodeGear/Embarcadero documentation for more information)

1.2.3.209 TTdbBlobProvider Properties

In *TTdbBlobProvider*

BlobIsEmbedded

BlobFormat

BlobSize

BlobFormatTag

BlobFormatName

Picture

BlobDataStream

LinkedBlobFileName

DataSource

FieldName

Derived from *TComponent*

(check Borland/Codegear/Embarcadero documentation for more information)

1.2.3.210 TTdbBlobProvider.BlobDataStream Property

Refers to a stream which can read and write blobs.

Delphi syntax:

```
property BlobDataStream: TStream read FBlobDataStream;
```

C++ syntax:

```
__property TStream BlobDataStream = {read=FBlobDataStream};
```

Description

Use *BlobDataStream* to read embedded blobs from a database table or write them into it.

1.2.3.211 TTdbBlobProvider.BlobFormat Property

Indicates the format of the current blob.

Delphi syntax:

```
property BlobFormat: TTdbBlobFormat read FBlobFormat;
```

C++ syntax:

```
__property TTdbBlobFormat BlobFormat = {read=FBlobFormat};
```

Description

Read this property to learn the kind of image that is currently active.

See also

TTdbBlobProvider.BlobFormatName

1.2.3.212 TTdbBlobProvider.BlobFormatName Property

Indicates the format name of the the current blob.

Delphi syntax:

```
property BlobFormatName: string read FBlobFormatName;
```

C++ syntax:

```
__property String BlobFormatName = {read=FBlobFormatName};
```

Description

Read this property to retrieve a human readable name for the image format that is currently active.

See also

TTdbBlobProvider.BlobFormat

1.2.3.213 TTdbBlobProvider.BlobFormatTag Property

Indicates a short name for the format of the current blob.

Delphi syntax:

```
property BlobFormatTag: string read FBlobFormatTag;
```

C++ syntax:

```
__property String BlobFormatTag = {read=FBlobFormatTag};
```

Description

Read this property to retrieve a short identifier for the image format that is currently active.

See also
TTdbBlobProvider.BlobFormatName

1.2.3.214 TTdbBlobProvider.BlobIsEmbedded Property

Indicates whether the current blob is embedded or linked.

Delphi syntax:

```
property BlobIsEmbedded: Boolean read GetBlobIsEmbedded;
```

C++ syntax:

```
__property bool BlobIsEmbedded = {read=GetBlobIsEmbedded};
```

1.2.3.215 TTdbBlobProvider.BlobSize Property

Indicates the size of the current blob in bytes.

Delphi syntax:

```
property BlobSize: Int64 read GetBlobSize;
```

C++ syntax:

```
__property int64 BlobSize = {read=GetBlobSize};
```

Description
Read this property to determine the number of bytes in the current blob.

1.2.3.216 TTdbBlobProvider.DeleteBlob

Deletes the current blob.

Delphi syntax:

```
procedure DeleteBlob;
```

C++ syntax:

```
virtual void __fastcall DeleteBlob(void);
```

Description
Call *DeleteBlob* to delete the current blob from the database.

1.2.3.217 TTdbBlobProvider.Create Constructor

Creates an instance of *TTdbBlobProvider* component.

Delphi syntax:

```
constructor Create(AOwner: TComponent); override;
```

C++ syntax:

```
virtual TTdbBlobProvider * __fastcall TTdbBlobProvider(Classes::TComponent * AOwner
```

Description
Call Create to instantiate a blob provider component at runtime.

1.2.3.218 TTdbBlobProvider.CreateTextualBitmap Class Method

Creates a bitmap displaying some text.

Delphi syntax:

```
class function CreateTextualBitmap(const Text: string): TGraphic;
```

C++ syntax:

```
virtual static TGraphic * __fastcall SetText(System::String Text);
```

Description

Use *CreateTextualBitmap* to create a bitmap showing some text that can be used to display a blob, which cannot be represented otherwise. For example if the blob holds a sound file or an image in an unknown format, the text can explain that fact to the user.

1.2.3.219 TTdbBlobProvider.DataSource Property

Defines the data source for the blobs.

Delphi syntax:

```
property DataSource: TDataSource read GetDataSource write SetDataSource;
```

C++ syntax:

```
__property TDataSource * DataSource = {read=GetDataSource, write=SetDataSource};
```

Description

The blob provider reads from and writes to the defined data source.

See also

FieldName property

1.2.3.220 TTdbBlobProvider.Destroy Destructor

Destroys the *TTdbBlobProvider* instance.

Delphi syntax:

```
destructor Destroy; override;
```

C++ syntax:

```
virtual void __fastcall ~TDataSet(void);
```

1.2.3.221 TTdbBlobProvider.FieldName Property

Defines the database column of the blob.

Delphi syntax:

```
property FieldName: string read GetFieldName write SetFieldName;
```

C++ syntax:

```
__property System::String FieldName = {read=GetFieldName, write=SetFieldName};
```

Description

This property determines in which field of the data source the blob is read and written.

See also

TTdbBlobProvider.DataSource

1.2.3.222 TTdbBlobProvider.LinkedBlobFileName Property

Indicates the file name in the case of a linked blob.

Delphi syntax:

```
property LinkedBlobFileName: string read FLinkedBlobFileName;
```

C++ syntax:

```
__property System::String LinkedBlobFileName = {read=FLinkedBlobFileName, write=FLi
```

Description

Read this property to learn the the file name of a linked blob.

See also

TTdbBlobProvider.DataSource

1.2.3.223 TTdbBlobProvider.LoadBlob Method

Loads the current blob.

Delphi syntax:

```
procedure LoadBlob;
```

C++ syntax:

```
virtual void __fastcall LoadBlob(void);
```

Description

Call *LoadBlob* to load the current blob and make it available through the *Picture* property. Usually this is happens automatically when the blob provider is connected to a data source and the data in the data source changes.

See also

Picture property

1.2.3.224 TTdbBlobProvider.OnReadGraphic Event

Occurs when a graphic is to be read from the database.

Delphi syntax:

```
TTdbBlobProviderReadGraphicEvent = procedure(Marker: Integer; var Graphic: TGraphic
```

```
property OnReadGraphic: TTdbBlobProviderReadGraphicEvent read
FOnReadGraphic write FOnReadGraphic;
```

C++ syntax:

```
typedef void __fastcall (__closure *TTdbBlobProviderReadGraphicEvent)(int Marker, T
```

```
__property TTdbBlobProviderReadGraphicEvent = {read=FOnReadGraphic,
write=FOnReadGraphic};
```

Description

Handle this event, if the application wants to create the graphic by itself instead of having the blob provider do it.

1.2.3.225 TTdbBlobProvider.OnUnknownFormat Event

Occurs when the blob provider encounters an image of an unregistered format.

Delphi syntax:

```
TTdbBlobProviderUnknownFormatEvent = procedure(Marker: Integer) of object;

property OnUnknownFormat: TTdbBlobProviderUnknownFormatEvent read FOnUnknownFormat
```

C++ syntax:

```
typedef void __fastcall (__closure *TTdbBlobProviderUnknownFormatEvent)(int Marker)

__property TTdbBlobProviderUnknownFormatEvent = {read=FOnUnknownFormat,
write=FOnUnknownFormat};
```

Description

Handle this event, if the application wants to handle the case of an unknown image format by itself.

1.2.3.226 TTdbBlobProvider.Picture Property

Provides the picture of the current blob.

Delphi syntax:

```
property Picture: TPicture read GetPicture write SetPicture;
```

C++ syntax:

```
__property TPicture * Picture = {read=GetPicture, write=SetPicture};
```

Description

The *LoadBlob* method loads the blob from the data source and creates a picture that is then available through the *Picture* property.

See also

LoadBlob method

1.2.3.227 TTdbBlobProvider.RegisterBlobFormat Class Method

Registers an image format.

Delphi syntax:

```
class procedure RegisterBlobFormat(BitMask, Pattern: Int64; const Tag,
Name: string; Format: TTdbBlobFormat; GraphicClass: TGraphicClass);
```

C++ syntax:

```
virtual static void __fastcall RegisterBlobFormat(int64 BitMask, int64
Pattern, System::String Tag, System::String Name, TGraphicClass *
GraphicClass);
```

Parameters

BitMask	64-bit mask to extract pattern bits from the first eight bytes of the blob data
Pattern	64-bit pattern to compare with the extracted pattern from the blob data
Tag	Short name for the format, e.g. *bmp*, *wav*, *wmf*, *gif*, *png*, *jpg*
Name	Long human readable name for the format, e.g. Windows meta file
Format	Numeric value identifying the blob format, must be unique within the table
GraphicClass	Class object of a *TGraphic* descendent which is able to *ReadData* and *WriteData* the format.

Description

By default, the blob provider can decode bitmaps, Windows meta files and wave files. Additional

image format like *gif*, *png*, *jpeg* etc. can be added by registering the format and a corresponding graphic class. When a blob is loaded, the blob provider ands the first four bytes of the blob with the bit mask and compares the result with the pattern. If it is equal, it creates an instance of the graphic class, passes a stream for the blob to it and calls the *ReadData* method. The resulting graphic is then handed over to the picture in the *Picture* property.

The blob provider reads the first eight bytes of the blob data as a 64 bit integer number, therefore the pattern bytes must by given in the opposite order of the physical sequence due to the little endian storage.

Example

This code registers GIF, JPEG and PNG formats with the blob provider.

```
ImageBlobProvider.RegisterBlobFormat($ffff, $d8ff, 'JPG', 'JPEG Image', tbfJPG, TJp
ImageBlobProvider.RegisterBlobFormat($ffffffff, $38464947, 'GIF', 'GIF Image', tbfG
ImageBlobProvider.RegisterBlobFormat($ffffffffffffffff, $0a1a0a0d474e5089, 'PNG', '
```

See also

LoadBlob method
Picture property

1.2.3.228 TTdbBlobProvider.SetBlobData Method

Writes an embedded blob.

Delphi syntax:

```
procedure SetBlobData(Stream: TStream; Format: TTdbBlobFormat);
```

C++ syntax:

```
virtual static void __fastcall SetBlobData(TStream * Stream,
TTdbBlobFormat Format);
```

Description

Use this method to write an embedded blob to the data source.

1.2.3.229 TTdbBlobProvider.SetBlobLinkedFile Method

Writes a linked blob.

Delphi syntax:

```
procedure SetBlobLinkedFile(const FileName: string);
```

C++ syntax:

```
virtual static void __fastcall SetBlobLinkedFile(const System::String
FileName);
```

Description

Use this method to write a linked blob to the data source. For a linked blob, only the file name is stored in the database and blob provider reads the the blob from the file, when it is loaded.

1.3 Database Engine

dataweb currently offers two database engines known as TurboDB Win, which runs on all 32-bit and 64-bit Windows platforms and TurboDB Managed, which runs on .NET Framework and .NET Compact Framework.

The TurboDB Engines are a very fast and compact database kernel that have proven to fit the needs of application programmers over the last eight years. They do not need any configuration, so they can be installed by just copying the program files. This feature makes them an ideal solution for downloadable, CD, DVD, mobile or Web applications, that are to run on a remote Web server only available via ftp.

This book contains all the features of the TurboDB database engines themselves, i.e. features independent of the development environment and the component library used. For information specific to the VCL component library for Delphi and C++ Builder, please refer to "TurboDB for VCL". The classes of the ADO.NET provider for the .NET framework are described in "TurboDB Managed".

Both TurboDB database engines have the following advantages:

- Small footprint
- Fast
- Runs without installation
- No configuration needed
- Supports multi-user access
- Royalty-free deployment
- Visual database manager included
- Many additional tools available for free
- Encrypted database files

The **managed engine** has the following additional advantages:

- It supports also Compact Framework, Silverlight and Windows Phone, so it runs also on mobile devices.
- Does not require any special rights for running in a .NET environment (e.g. unsafe code)
- Supports user-defined functions, stored procedures and user-defined aggregates.

The **native engine** has these additional advantages:

- Native 32-bit and 64-bit code
- Special column types for one-to-many and many-to-many relationship between tables

1.3.1 New Features and Upgrade

1.3.1.1 New in TurboDB Win32 v6

TurboDB 6 comes with a large number of new and enhanced features. Note that some of those new features require existing tables to be updated to table level 6 in order to be available. Check in the respective section to learn whether this is the case.

- Faster handling of Unicode strings in all areas of the database engine
- Faster calculations in all areas of the database engine
- Optimized index storage format for index entries with variable size. Especially indexes on larger string fields profit by less memory requirements and faster execution.
- Precise result data types for calculated columns in SQL
- String columns now have collations that define, how strings from that column are sorted and compared.
- File extensions for database objects like tables, indexes etc. have been changed for the new table level 6. They now all start with *tdb*, to avoid naming conflicts with other applications.
- User-defined separators for full-text indexing. See the CREATE FULLTEXTINDEX SQL command.
- Selective full-text search in SQL: The CONTAINS predicate now also supports definition of specific columns, in which the search words must be appear:

WHERE CONTAINS('aWord' in Column1, Column2, Column8)

- Calculated columns can be defined in the *ALTER TABLE* and *CREATE TABLE* command.

See also

Upgrade to TurboDB Win v6

1.3.1.2 Upgrade to TurboDB Win v6

When upgrading from TurboDB 5 to TurboDB 6 you must take care of the following issues:

Language Drivers

TurboDB 6 does not support language drivers anymore. Instead, tables and columns can have collations, which define the sorting of strings. When you want to upgrade a table to TurboDB 6 which uses a language driver, you must remove the language driver using TurboDB 5, upgrade the tables to level 6 using TurboDB 6 and assign the desired collations.

String Sorting and Comparison

Strings are now sorted and compared according to their collation. Since sorting and comparison were different in TurboDB 5, comparison of strings in older tables is now case insensitive. If that does not fit your requirements, assign another collation.

Important Hint

Please note, that TurboDB 6 cannot share TurboDB databases with TurboDB 5 at the same time. You will receive an error message saying the locking files are incompatible. If there is no TurboDB 5 application currently running, but there are still net/rnt/mov/rmv files around, the same error will shine up. Delete the files and everything will work as expected.

When upgrading from TurboDB 4 to TurboDB 5 some points have to be considered:

Important Hint

Please note, that TurboDB 5 cannot share TurboDB databases with TurboDB 4 at the same time. You will receive an error message "Error in log-in". If there is no TurboDB 4 application currently running, but there are still net/rnt/mov/rmv files around, the same error will shine up. Delete the files and everything will work as expected.

New Reserved Keywords

The following identifiers are new reserved keywords in TurboDB 5. If your database schema is using those words as table or column names, you must either enclose them in double quotes wherever they appear in SQL statements or use different names:

ACTION, ALL, ANY, CASCADE, CASE, CAST, DICTIONARY, ENCRYPTION, EXCEPT, EXISTS, FULLTEXTINDEX, INTERSECT, NO, SOME, TOP, UNION, WHEN

Use of Quotes

TurboDB 5 uses single-quotes to denote string literals exclusively. Double quotes were allowed in TurboDB 4 as well but must be changed when upgrading. Double quotes now exclusively denote identifiers and can be used to work with names that contain spaces or are identical to reserved keywords.

Encryption

TurboDB now supports additional methods for strong encryption. Due to this fact the syntax of the *create table* statement and the *alter table* statement has been modified.

Full-Text Indexes

Full-text indexes have been thoroughly re-designed to be faster and maintained. The old full-text indexes are still supported for the older table levels but if you want to upgrade your tables you have to re-write the full-text part of your SQL code.

1.3.1.3 New in TurboDB Managed v2

Version 2.0

- The new CONTAINS predicate together with the new full-text indexes allow for powerful and very fast search for arbitrary words.
- Table names can have up to 79 characters and column names up to 128 characters.
- Connections are now pooled for better performance in applications that execute multiple commands and close the connection in between.
- Fast encryption and Rijndael (AES) encryption is now supported.
- Programmatic compression of databases is provided by the *TurboDBConnection* class.
- UNION, INTERSECT and EXCEPT statements are implemented.
- TurboDB Pilot has a much more comfortable management of databases and commands.
- TurboDB Pilot now provides a visual table designer for creating and altering database tables.

Version 1.3

TurboDB Managed 1.3 contains a completely new family of features centered around programmability.

- User-defined functions allow you to define your own SQL functions either in SQL or as a .NET assembly.
- Stored procedures allow you to call complex statement sequences with a single call. They are implemented either in TurboSQL or in any .NET language.
- User-defined aggregates allow you to define new aggregation function for your SQL statements in any .NET language.

Read the section on "Programming Language" for detailed information.

1.3.1.4 Upgrade to TurboDB Managed v2

TurboDB Managed 2.0 is fully compatible with version 1.x.

New reserved keywords

There are however some new keywords. If these identifiers are used as table, column or routine names, they have to be quoted with brackets [...] or double quotes "..." in statements: BY, CORRESPONDING, ENCRYPTION, EXCEPT, INTERSECT, UNION, CONTAINS, FULLTEXTINDEX.

1.3.2 TurboDB Engine Concepts

This chapter explains basic concepts of the TurboDB database engines.

Overview describes general features like limits and naming.

Databases explains the difference between single-file databases and directory databases.

Indexes explains the different kinds of indexes TurboDB supports.

Automatic Linking presents the TurboDB concept for quicker and less error-prone data modeling.

Multi-User Access and Locking describes, how TurboDB implements the multi-user access.

Optimization lists items you can check, when your application is not fast enough.

Error Handling describes, how errors from the database are reported and handled.

Miscellaneous talks about the physical database files, data security and localization.

1.3.2.1 Overview

Limits

Table and Column Names

Column Data Types

1.3.2.1.1 Compatibility

There exist two implementations of the TurboDB database engine, the native Windows engine and the .NET engine also called the managed engine. The two database engines can work with the same database files as long as the restrictions of the two engines are observed. These are the rules to follow, when you want to share a database file between TurboDB Managed 2.x and TurboDB Win 5.x.

- Use a single-file database instead of a directory database.
- Restrict table and column names to 40 characters.
- Do not use Blowfish encryption, but employ *FastEncrypt* or *Rijndael*. The password for all tables must be the same as TurboDB Managed only supports one password per database.
- TurboDB Win will ignore user-defined functions and stored procedures. They cannot be used in computed indexes and queries.
- Do not use language drivers.
- TurboDB Managed and TurboDB Win cannot share the same database file concurrently since the locking system is different. They can however access the data alternately.

1.3.2.1.2 System Requirements

TurboDB Managed

.NET Framework 2.0 or higher

TurboDB Win

Operating system: Windows 32 bit since Windows 2000 and Windows 64 bit since Windows Vista - all editions supported by Microsoft without Windows Phone/Windows RT.

There are 64-bit and 32-bit versions of TurboDB Win. Also the 32-bit version runs on the 64 bit editions of Windows in the 32 bit mode without problems.

1.3.2.1.3 Limits

Some technical data valid for tables level 4 and above:

Maximum number of records per table	2 G
Maximum size of a data table	4 EB (one exabyte is 1 G times 1 GB)
Maximum number of columns per table	1000

Maximum size of one table row	32 KB
Maximum number of user-defined indexes per table	48 (table level 4 and above) or 14 (up to table level 3)
Maximum number of levels in a hierarchical index	255
Maximum size of the calculated index information	32 KB
Maximum number of tables per database	255
Maximum length of a string column	32.767 characters
Maximum number of link columns per table	9
Maximum total size of all blobs of a table	1 TB (block size 512 B) to 64 TB (block size 32 KB)

1.3.2.1.4 Table and Column Names

Identifiers

Identifiers consist of a true character followed by alphanumeric characters, the underscore _ and the hyphen. They can contain up to 40 characters. German umlauts count as true characters.

Valid identifiers are:

```
Street
Avarage_Duration
Date-of-birth
Address8
Lösung
```

Column and Table Names

Column and table names can contain any ANSI characters but the control characters, the brackets [] and the double quotes. The maximum length is 40 and it must start with a real character. If the name follows the rules for an identifier, it can be used normally in all expressions and statements. If it does not, it must be included either in double quotes or in brackets.

These are examples of **invalid** identifiers, which can be used as column names if enclosed in double quotes or brackets:

`No of Items` (spaces not allowed)
`3645` (first character must be non-digit)

1.3.2.1.5 Column Data Types

TurboDB offers the following types of table columns. This list refers to the data types of the storage engine, their SQL counterparts are described in "TurboSQL Column Types".

String

A string field holds alphanumeric characters up to the given limit. The maximum size is 32765 characters (255 for table level 3 and below). A string field holds one byte for each character plus one or two bytes for the length of the string plus an additional byte, if the string is nullable.

WideString

Up to 16382 Unicode characters (255 for table level 3 and below). The actual field size in bytes is twice the number of characters plus two for the string length plus one, if the string is nullable.

Byte

Numbers from 0 to 255. Byte fields can have an optional null indicator. The size is one or two bytes depending on the null indicator.

SmallInt

Numbers from -32767 to +32768. SmallInt fields can have an optional null indicator. The size is two or three bytes.

Integer

Numbers from -2147483648 to +2147483647. Integer fields can have an optional null indicator. It takes four or five bytes in the database table.

LargeInt

Numbers from –2^63 to +2^63 – 1 with an optional null indicator. One Int64 field uses eight or nine bytes in the table.

Float

Holds a 8 byte floating number, i.e. from 5.0e-324 to 1.7 x 10e308. Can have an optional null indicator. The size is eight or nine bytes.

Time

Values from 12:00:00.000 am to 11:59:59.999 pm. Precision is either minutes, seconds or milliseconds and must be given when creating a level 4 table and above. In level 3 tables and below, precision is always minutes. Can have an optional null indicator. Depending on precision and null indicator, size is between two and five bytes.

Date

Values from 1/1/0000 to 12/31/9999. Size is four bytes. Internally dates are represented as a packed bit field.

DateTime

Values from 1/1/0000 12:00:00.000 am to 12/31/9999 11:59:59.999 pm. Values take eight bytes internally.

Boolean

Holds a Boolean value: *True* or *False*. Can have an optional null indicator. Size is one or two bytes.

Enum

Holds one of a definable set of named values, e.g. *mon*, *tue*, *wed*, *thu*, *fri*, *sat*, *sun*. The values have to be valid identifiers similar to column names. They can be converted to textual representation using the function *Str*. Example for a enumeration field *gender* with values male, female, unknown: When the value female has been assigned *Str(gender)* returns the string 'female' while *gender* by itself returns the number 2.
An enumeration value may have up to 20 characters, the maximum number of enumeration values is 15 and the total length of all enumeration values including separators must not exceed 255 characters.

Memo

Long strings of variable length up to 1 GB. Memos are stored in additional storage objects called the memo file (extension *mmo/tdbm*).

WideMemo

Unicode string of variable length up to 1 GB. Wide memos are stored in the blob storage object (extension *blb/tdbb*).

Blob

Images and other binary data of variable length up to 1 GB. Blobs are stored in an additional storage object (extension *blb/tdbb*).

Link

Pointer to another record in the same or another table (1:n relation). The target table is fixed for all link values of this column. Links are explained in "Automatic Linking". A link field contains the record id of the record the field is linked to. Its size is four bytes.

Relation

Pointer list to other records in the same or another table (m:n relation). The target table is fixed for all link values of this column. Relations are explained in "Automatic Linking". Relations fields are not physical columns in the database table itself. There is an additional database table created

transparently for each relation field that holds one link per row. The relation table has two link columns, one of which points to the database table that has the relation field and the other one points to the database table the relation field links to. Such the relation field itself has a size of zero bytes.
Feature is not supported in TurboDB Managed

AutoInc

Counter that gives a unique number to each record. *AutoInc* fields are assigned its values by the database engine and can not be edited. The Automatic Data Link mechanism uses AutoInc fields as the primary index to store record references. An *AutoInc* column has an optional *indication* property, which can be set to a list of column names. This property is used for Automatic Linking and can be left empty to make the *AutoInc* column behave just like a "normal" one.

GUID

128 bit number used by MS COM and ActiveX technologies. *GUID* stands for Globally Unique Identifier. *GUID*s are usually calculated by a OS function which assures that no other call anywhere on earth will produce the same value.

Hint

Using data types AutoInc, Link and Relation will automatically generate one ore more indexes. Depending on the table level these (system) indexes are named as the table or start with prefix 'sys_'. Modifying or deleting of these indexes is not possible.

See also

TurboSQL Column Types

1.3.2.1.6 Collations

Collations define how strings are sorted and compared. TurboDB uses a similar collation naming schema as Microsoft SQL Server. A collation name consists of a Windows locale name plus two or four characters that indicate, whether the string is case sensitive and diacritics sensitive. The meaning is:

- *AS*: Sensitive for diacritics (accents)
- *AI*: Insensitive to diacritics (accents)
- *CS*: Case sensitive
- *CI*: Case insensitive

If both are given the case specification must precede the diacritics specification. Each specification can be omitted, in which case the diacritics sensitivity defaults to true and the case sensitivity defaults to false. In addition to the Windows locale names, the special collation *TurboDB* indicates the sorting of TurboDB 5 and before. Therefore you cannot append the *as*/*ai*/*cs*/*ci* specifications to the collation name *TurboDB*. Collation names themselves are case insensitive.

The following are some examples of valid TurboDB collation names:

- *German* (diacritics sensitive, case insensitive)
- *GERMAN* (same as above)
- *English_ai* (diacritics insensitive, case insensitive)
- *Spanish_ci_as* (diacritics sensitive, case insensitive)
- *Russian_cs* (diacritics sensitive, case sensitive)
- *Russian_CS* (same as above)
- *TurboDB* (diacritics sensitive, case insensitive)

These are examples for **invalid** collation names:

- *Collation1_ai* (Collation1 is not a Windows locale)
- *Spanish_as_ci* (Wrong order of sensitivity specifications)
- *Spanish_ca* (Invalid sensitivity specification)
- *TurboDB_cs* (Special collation *TurboDB* cannot have additional specifications)

Tip: Use the table properties window of TurboDB Viewer to display a list of collations available on your system.

Collations can be assigned and checked in different ways:

- TurboDB Viewer displays collations and allows to choose one, when creating or altering a table.
- TurboSQL supports the COLLATE clause for column data types.
- In the VCL library the *TTdbFieldDef* class has a property named *Specification*, which takes the collation name for string types.
- The .NET providers support collation names through the appropriate ADO.NET interfaces.

Compatibility

Collations are supported as of TurboDB Win v6 and TurboDB Managed v3. Only tables of level 6 and higher allow the collation definition on a per column basis. Tables of a lower level allow the collation definition on a per table basis using the three letter ISO code for the language. Earlier versions of TurboDB used language drivers, which a no more supported. See the upgrade instructions to learn how databases can be migrated.

Because string comparison is now 100% consistent in filters, SQL, TurboPL and indexes, the comparison in older TurboDB tables is now case insensitive. That means that *MyStringColumn = 'TestString'* is true for a *MyStringColumn* value of *'teststring'* in TurboDB Win 6, whereas it was false in TurboDB Win 5 and below. If you want the case sensitive comparison, define another collation for the table or for the column.

See also

CREATE TABLE statement

1.3.2.2 Databases

TurboDB supports two different types of databases.

Single-File vs. Directory Database

Directory Database

Directory databases are folders on a hard disk where all TurboDB database objects reside in different files. Database directories have been supported ever since TurboDB exists.

Single-File Database

A single-file database is one single file which contains all database objects of the database. Such a database file has the default extension of *.tdbd. Single-file databases are supported as of version 4.0. Single-file databases have the advantage of being very easy to deploy or just to copy and move around on your hard disk. The advantage of database directories is, that they are slightly faster and that you can share tables between different databases.

Single-file databases are implemented using a virtual file system layer by dataweb. This layer maps the database objects either to different files in a database directory or to a single database file. dataweb offers a tool - the dataweb Compound File Explorer - which can open such database files and show the content. You may also move database objects from and to the database file. This is a way to convert a directory-based database to a single-file database or vice versa as well.

While TurboDB Native supports both of these database types, TurboDB Managed can only work with single-file databases.

Databases with Catalogs

In previous versions TurboDB databases were just a collection of database tables stored in separate files. While this approach has the advantage of being able to share database tables between databases, it also has some disadvantages. One disadvantage is, that a table name cannot always be resolved, because the table file may be located in a folder far away. Another one is, that multiple passwords are required to open the database, if the tables are encrypted in different ways.

For this reason TurboDB Win 5 introduced databases with catalogs, which store a list of tables and additional database-wide properties. Catalogs are most useful for dictionary databases. With singe-file databases, the above disadvantages do either not exist or are less problematic.

TurboDB Managed supports only single-file databases and keeps track of all necessary information by itself. It does therefore not offer an explicit catalog support.

Note: Databases with catalogs were called managed databases in previous versions but the term conflicts with TurboDB Managed .

1.3.2.2.1 Sessions and Threads

As of TurboDB version 4 you need to create a session before you can open tables and queries. If you are using a component library however (e.g. TurboDB for VCL) session handles are hidden within a database or connection object.

You may create as many sessions as you want, but you should be aware of some consequences in a multi-session application:

- Cursors of the same table within different sessions are synchronized on file level. This is much slower than the synchronization of cursors within the same session, which is performed in the memory.
- You can use different threads for different sessions, but you should not use different threads on the same session. For performance reasons there is no built-in thread synchronization within the same session.

1.3.2.2.2 Table Levels

Along with the enhancements and improvements in TurboDB, different storage formats have been developed to support additional features. They are called table levels and the following will describe the characteristics of each. While TurboDB Native supports all these table levels, TurboDB Managed can only work with table level 5.

Table Level 6

- Uses file extensions in the tdb? schema for all files.
- Names relation tables after the main table they belong to.
- Supports Windows collations on table and column level.
- Offers full-text indexes with configurable separators.
- Has index structures specifically for string indexes.
- Supports encryption for indexes.

Table Level 5

- Is compatible with TurboDB Managed.
- Supports standard SQL syntax for checks, default values and calculations for columns and indexes.
- Supports tables with the same name in different database files in the same directory.
- Includes the table schema in the encryption.

- Is prepared for character set support.
- Is prepared for Unicode table and column names.

Table Level 4

- Supports primary keys and unique keys.
- Supports time columns with a precision of seconds or milliseconds.
- Supports additional (strong) encryption algorithms.
- Supports checks and foreign keys.
- Adjusts the ordering to SQL standard, such that null values are less than all other values.
- Increases the number of indexes allowed for a table.
- Supports strong encryption.
- Allows for maintained full-text indexes.

Table Level 3

- Adds Unicode strings, date time columns and GUID columns.

Table Level 2

- Introduces 32 bit Integers and Ansi encoded strings.

Table Level 1

- The original table file format. Compatible with TurboDB for DOS.

1.3.2.3 Indexes

Indexes are additional storage objects for a database table that enable fast searching and sorting. TurboDB indexes are built on either a list of field names or an expression to define the sorting order. If an index is declared to be unique, records that would create a duplicate key in the index are not accepted. Another kind of indexes are full-text indexes.

Indexes Based on a Field List

These indexes are sorted in the order of the first field in the field list. If two records have the same value for the first field they are sorted after the second field of the field list and so on. There can be up to 10 fields in the index field list. Every field can be sorted in ascending or in descending order.

Indexes Based on an Expression

These indexes are sorted after the value of an arbitrary expression that can be up to 40 characters long. If the expression is of string type, the index is sorted like if the expression values were values of a string column. If the expression is of numeric type, the index is sorted according to normal numeric order.

Full-Text Indexes

A full-text index enables the user to search for a keyword or a set of keywords in any field of the table. Full-text indexes require a separate table, the dictionary table, which contains the indexed words. Full-text indexes are implemented differently for table level 4 and the levels below. In table level 4, there is only one storage object to make-up the connection between the dictionary and the table, it has the extension *fti or tdbf*. In the older table levels, the connection was implemented using relation fields, which requires an additional base table (extension rel) and two indexes (extension in1 and in2).

System Indexes

Using data types AutoInc, Link and Relation will automatically generate one ore more indexes. Depending on the table level these (system) indexes are named as the table or start with prefix 'sys_'. Modifying or deleting of these indexes is not possible.

Indexes can be created and deleted with various TurboDB tools at design time. At run-time, you

can use TurboSQL to create, update and delete indexes. Also some component libraries (e.g. VCL) contain methods for adding and deleting indexes.

1.3.2.4 Automatic Linking

Very often tables are linked the same way in all queries. E.g. items are linked to the invoice they belong to, authors are linked to the books they have written and so on. Therefore TurboDB allows you to specify different links from one table to other tables in the table itself.

Imagine you have an invoice table where the records contain the date of the invoice, the customer no, the invoice no and other invoice-related information. The items are in another table that has columns like *article no*, *price*, *total amount* and others. How do you link the item to the corresponding invoice it is part of? The traditional way is to have an additional column in the item table that designates the invoice no of the invoice the item belongs to. Every query that respects the invoice-item relation has to contain the following condition: ...where "ITEM.invoice no" = INVOICE.no...

What is it?

Even if you can still do this the traditional way with TurboDB, the preferred way of doing it is a little different. Rather than having an *invoice no* in the ITEM table you would use a pointer to the INVOICE table called link field. Because the default in TurboDB is to have a (unique) record id in every table the link column in the item table just holds the record id of the invoice it belongs to. Because the definition of the link column contains the information that the values in this column point to table INVOICE, the database now knows about this relation and will by default assume it in every query. This way of linking tables has some great advantages:

Doing queries with linked tables is easy because the system "knows" how the tables have to be linked. Queries can be much faster, because a record id is just a number while secondary keys often are much more complex. Changing indexes, column names or types does not affect the link at all. It is very easy to access the record of the master table with a special link notation.

You can look at link fields as an object-oriented way to work with database tables. They do not strictly conform to the relational paradigm but bring the feeling of pointers and references into the game. The item "knows" to which invoice it belongs. This link is given by the nature of things and will not probably change very often.

Another advantage is that link and relation fields need not display the purely technical AutoInc values to the user. If you assign an indication to the AutoInc column, link and relation columns will display this information instead of the numerical one. Here is an example:

CREATE TABLE DEPARTMENT (Name CHAR(20), Id AUTOINC(Name)

CREATE TABLE EMPLOYEE (LastName CHAR(40), Department LINK(DEPARTMENT))

The query

SELECT * FROM EMPLOYEE

will display a list of last names and department names, because the department name is defined as the indication for the AutoInc column.

How is it done?

While link columns introduce easy 1:n relations (one invoice has many items), this object-oriented concept makes as ask for a m:n relation i.e. a list of pointers in one table pointing to another table. TurboDB relation fields are the answer to this. A table containing a relation field to another table links every record to a number of records in the other table and vice versa. Taking again books and authors as example, inserting a relation column in the BOOK table would take care of the fact that a book can be written be more than one author and that one author might contribute to more than one book.

As you might suspect, relation fields are not so easy to implement as link fields. M:n relations have to be realized by an additional table in between that has one record for every link between the

tables. This is exactly what Turbo Database does when you define a relation column for your table pointing for example to table AUTHOR. TurboDB will create a hidden intermediate table containing a link column to table AUTHOR and another link column to table BOOK. This is what you had to do if you worked in the traditional way. But with TurboDB the intermediate table is created and maintained automatically and transparently to you.

Compatibility Information

This feature is only partly supported in TurboDB Managed. TurboDB Managed currently supports link columns but not relation columns.

1.3.2.4.1 Working with Link and Relation Fields

In order to profit from automatic linking you should think of adding link and relations columns to every table you create. You will soon find it very natural to add the linking information into the table. After all you do the same with your Delphi, C++ and/or Java classes, don't you?

Adding Link and Relation Columns

When you want to work with link and relations fields to establish a one-to-many or many-to-many relationship between tables, you must decide which of the tables is the source and which is the target of the relationship. The first is called the child table and the second the parent table. It is just like the referencing table and the referenced table when you are working with traditional foreign keys.

The parent table must include an AutoInc column, which is used at the primary key for the linking. The child table must include a link or relation column to establish the relationship. The link column can store exactly one pointer to the parent table. The pointer is displayed as the AutoInc value in the parent table or as the column values of the indication, if you have defined one with the AutoInc column of the parent table. The relation column stores multiple pointers to the parent table, which are displayed as a list of AutoInc values or column values according to the indication definition for the AutoInc column.

Link and Relation Columns with Direct Table Access

(Direct table access is a feature available for the VCL component library but not with ADO.NET.)

Once you have defined your links and relations, they are respected by the database in every query. Even if you don't have any search-criteria, only corresponding detail records will be shown for every master record. In the rare case that you don't want this default linking you may always enter another equate join that overrides it.

Link and Relation Columns with TurboSQL

In TurboSQL queries the relationships defined through link and relation fields are not created automatically. Use a simple JOIN to utilize the reference:

```
SELECT * FROM Master JOIN Detail ON Detail.LinkField = Master.RecordId
```

For adding new rows to the tables, the function *CurrentRecordId* should be used:

```
INSERT INTO Master VALUES(...); INSERT INTO Detail VALUES(...,
CurrentRecordId(Master), ...)
```

This compound statement inserts first a record in the Master table and then a record into the Detail table while using the last value for the record id in the master table as the new value for the link field in the detail table. Therefore the detail record is linked to the master record.

Compatibility Information

This feature is only partly supported in TurboDB Managed. TurboDB Managed currently supports link columns but not relation columns.

1.3.2.5 Transactions

TurboDB supports transactions based on an additional storage object per table, the redo-log. When an transaction is started, all subsequent modification to the database tables will be entered in the redo-logs after they have been materialized to disk. If the transaction is committed, the redo-log is simply deleted. If the transaction is rolled-back, the information from the redo-log is used to undo the modifications. This means, TurboDB follows an optimistic transaction schema: Committing a transaction is very fast, while undoing it, requires much more work.

The tables, which have been modified by the transaction are locked to other sessions until the transaction is finished. For performance reasons, tables, which have been read during the transaction are not locked. Therefore the transaction level in TurboDB is read-committed.

Because TurboDB clients can interact on file level (i.e. without a database server), the handling of clients that die during a transaction is more difficult. TurboDB engine can detect the fact, that another client has died and performs the roll-back. Because in the scenario, another client undoes the modifications of the died client, we call this mechanism hijacking.

1.3.2.6 Optimization

When you have the feeling that your TurboDB application is slower than it should be, there are many possible reasons. Check the following questions and follow the appropriate instructions.

One or more select statements on a large table or on a set of tables takes too long to execute

There are basically two ways to speed a query: Create the necessary indexes and/or optimize the statement.

Setting a filter on a table component (VCL library) takes too long

Adding an index to your table may also help in this case. Another way is to use the range feature instead of the filter.

In file access mode local operation is quite fast, but as soon as a second application just connects to the database, everything slows down

The first thing to check in this case, is whether your network has problems. Because file access mode makes use of network functionality very heavily, it often happens that poor network performance was not noticed before the TurboDB application was installed.

The network is ok, but when a lot of people edit in the database, completing an operation takes very long

With a lot of people working concurrently on the database, the locking overhead grows and a lot of waiting for access to the database occurs. The first step in this case is to use explicit locking to form bigger database operations and thus less locking overhead. If this won't help enough, you can still use TurboDB Server.

In addition to those specific hints, there some general hints that can help increase database performance:

Set the flush mode to fast

The flush mode determines the buffering degree within the database. Setting it to fast will maximize the internal buffering and therefore increase performance. However, if the application crashes, data loss can occur in this mode.

Use exclusive access if possible

In case your application is conceived for a single-user only. You should put the database in exclusive mode to eliminate multi-user access overhead.

1.3.2.6.1 Network Througput and Latency

The network performance is critical for the performance of your TurboDB applications especially, when it is used in multi-user mode. Network problems are the most frequent reason for poor database throughput and should therefore checked out first in case of performance problems. You might have a network problem, if the computer with the database files is different from the one with the application and your application is generally very slow or gets generally very slow, as soon as a second database client is started.

Because network problems are sometimes hard to detect, dataweb offers a free program, which measures network operations typical for databases called *NetTest*. You can download this program from our Web site or request it from the dataweb support. This test program will enable you to determine within five minutes, whether the network is the cause of a performance problem or not.

In case the network turns out to be slow, there are different points to check out:

- Check whether some virus checker or other software is conflicting with TurboDB files. These programs tend to check these highly dynamic files after each modification and thereby slow down or even completely block the database access. You can safely configure the virus checker to not check TurboDB files, because they are not executable.
- Check whether there is updated driver software for your network adapter or if it is defect.
- There is a known problem with SMB signing, if you access a Windows 2000 domain controller from Windows XP clients. Please refer to Microsoft knowledge base article 321098 for more information and the resolution.
- Hubs between database clients and the database file server sometimes block access, if the network traffic is too high. In this case they should be replaced by a good switch.

1.3.2.6.2 Secondary Indexes

Additional indexes for database tables can accelerate queries and filters by some orders of magnitude. Consider a simple query like:

```
select * from Customers where State = 'NJ'
```

or the similar filter condition

```
State = 'NJ'
```

Without an index, TurboDB has to scan every record in the table to select those, which satisfy the condition. And while TurboDB is optimizing multiple reads quite well, the operation nevertheless can take up to some minutes, if the table is large (a few million records).

If however there is an index, which starts with the State column, calculation the result of this selection is instantaneous, because TurboDB will be able to directly sort out the correct rows.

Also with joins, an additional index can do wonders. Look a this query:

```
select * from Customers, Orders where Orders.CustNo = Customers.No
```

or the equivalent

```
select * from Customers join Orders on Orders.CustNo = Customers.No
```

Also in this case, an index over *Orders.CustNo* or *Customers.No* will speed the query considerably. Depending on which one exists, TurboDB will execute the statement such that it can be used. However, since *Orders* will be a much larger table than *Customers* (the average number or orders per customer should be greater than one), an index over the *Orders.CustNo* field will bring you more in terms of execution time gain than an index over *Customers.No*. (The latter will most probably exist nevertheless, because *No* tends to be the primary key for the *Customers* table.)

The disadvantage with indexes is that their maintenance takes time during the change operations delete, insert and update, which must be taken into account as well, when regarding the overall performance of your application. Because in most applications queries are much more frequent

than changes, indexes for crucial use cases will pay off most of the time.

1.3.2.6.3 TurboSQL Statements

Some rules for fast TurboSQL statement execution:

Start the where and the having clause with simple and-ed conditions

If logically applicable, order your search-conditions like this:

```
A.a = B.a and C.b > X and (...)
```

i.e. start with simple comparisons, which are necessary for the whole search-condition to be satisfied. These simple comparisons are most suited for optimization. The optimizer will try to create this structure of the search-condition automatically but may in some cases not be smart enough to do so.

Separate the column-reference from the value in comparisons

If you write

```
A.a > 2 * :ParamValue,
```

this will be optimized more likely than

```
A.a/2 > :ParamValue.
```

The important point here is that the column reference *A.a* stands alone the left side of the comparison.

Prefer *like* over Upper

The condition

```
A.a like 'USA'
```

can be optimized, while

```
Upper(A.a) = 'USA'
```

cannot.

Prefer left outer joins over right outer joins

The implementation of joins largely favors left outer joins. Whenever it is suitable in your application, write

```
B left outer join A on B.a = A.a
```

instead of

```
A right outer join B on A.a = B.a.
```

This can speed up your statement considerably. The optimizer does not do this conversion by itself, because it would deprive you of the possibility to hand-optimize your statement.

Modify the Sequence of Tables in the From Clause

This sequence can have a severe impact on the query performance. If you think, your query is not as fast as it should be, just check out different orderings in the table-reference list.

```
select * from A, B, C
where ...
```

might be much faster than

```
select * from C, B, A
where ...
```

Normally the optimizer will try to order table-references not part of a join in the best way, however sometimes assistance from the programmer is needed.

1.3.2.7 Miscellaneous

Database Files describes, which physical database files exist in TurboDB and what they are good for.

Data Security explains the various methods to secure your data.

Localization outlines the way, how you can adopt your TurboDB application to special locales.

1.3.2.7.1 Database Files

This topic explains the files that make up a TurboDB database. They are named after the file extension used. If you are working with a single-file database, you cannot see these file names directly. But when using the dataweb Compound File Explorer, you will see that the database file contains storage objects (also sometimes called files) with the same name extensions.

Level 6	Level 1-5	Description
tdbd	*tdbd*	*tdbd* stands for TurboDB database. Such a file contains all tables and indexes, which belong to a database, if the database was created as a single-file database. In this case there are no real *dat*, *mmo*, *blb*, *rel*, id, in? and *ind* files seen in the file system, because the respective data is stored within the *tdbd* file.
tdbt	*dat, rel*	Contain the database tables, that is the records. Rel files are special database tables created transparently to implement many-to-many relationship. Deploy with your application.
tdbm, *tdbb*	*mmo*/*blb*	These are the memo and blob files, that exist once for each table that has at least one memo field or at least on blob field. One such file contains all the data of all the memo or blob fields in the table. Deploy with your application.
tdbi	*ind*	User defined index. Each *ind* file contains one index. Deploy with your application.
tdbi	*id*, *inr*, *in?*	Automatically generated indexes for tables of level 3 and below. The inr file is an index on the AutoInc field and the id file is an index on its indication. The in? indexes (in0, in1 etc) index link columns. Deploy with your application.
tdbf	*fti*	Full-text index
tdbl, *tdbv*	*net*, *rnt*, *mov*, *rmv*	These are the lock files and exist for each table open in shared mode. Do not deploy with your application since these files contain only dynamic information. When your application crashes or is reset during debugging these files happen to remain on your hard disk and will lead to error messages like "table is in use by another application". In this case, just connect to the database with TurboDB Viewer or any other TurboDB application, view the corresponding tables and the files will be deleted when you close this application.
tdbr	*tra*, *rtr*	These files are the redo log files for transactions. During an transaction, there is one tra file (rtr for relation tables) for each table modified during the transaction. When the transaction is finished, those files are deleted. If you see those files with your database, when no application is currently accessing it, this means, an application crashed during a transaction. Do not delete the redo logs then, the application that accesses the database tables next, will rollback the interrupted transaction to restore database integrity.
		Temporary tables have random file names like jzbgopqw.dat and the temporary indexes are called appropriately. These files are usually stored in the user's temporary directory. But you can define any other directory using the PrivateDir property in one of the library components.

1.3.2.7.2 Data Security

Normally your TurboDB database tables can be opened by any person who has access to the file and who uses a tool that can read TurboDB database files. To prevent people from doing so, you can define a password for your tables. All TurboDB tools respect this password and will not show the content of the table unless the user has entered the correct password.
While this is a very useful feature in many cases, it is not a true protection for your data, because one can still read the content of a database or table file with any binary editor or even a text editor. This is also true if you assigned a password to the table, because the password does not change the way database values are stored. If you want to secure your data from being viewed by unauthorized people, TurboDB Engine offers a variety of encryption algorithms, which encrypt every record when it is written to the file.

The classic TurboDB encryption algorithm is based on a 32-bit key. As you might know, a 32-bit key in our days is not secure enough to do banking or other high security things. But for most applications this level of security is appropriate and a shorter key speeds up database transactions.

If you need strong encryption for your data, you can use one of the strong encryption algorithms offered in TurboDB. With these algorithms in place, your data is secured from anybody, who does not know the key. With the current state of encryption technology, these ciphers cannot be broken even with sophisticated decryption algorithms and computer hardware.

The encryption method can be defined on database level (for managed databases) or on table level. If you define the encryption on database level, you have to define the encryption method and the password only once when creating the database. And the user must enter the password only once for all tables in the database. Therefore, this is the recommended way.

In previous versions of TurboDB, encryption required both a password and a 32-bit number called the code to connect to a table. Current versions only require one string, the password. For compatibility, the former combination of password and code is now merged into one string like this <password>;<code>. For example the password *secret* and the code *-3871* are now entered as the password *secret;-3871*.

This is a list of all available security options. They are indicated as the encryption method enumeration in the different libraries.

Name	Description	Key	Compatibility
Default	For tables in managed databases: Takes the encryption parameters from the database. Other tables and databases: No encryption	See respective row	-
None	Neither encryption nor protection	-	-
Protection	The table is not encrypted but requires a password to be opened	The password, e.g. 3Huv	All table levels. All versions of TurboDB Win. TurboDB Managed 2.x and above.
Classic	The table is encrypted with fast encryption and has an additional password.	The password and the numeric encryption code separated by a semicolon, e.g. 3Huv;97809878	All table levels. All versions of TurboDB Win. TurboDB Managed 2.x and above.
Fast	The table is encrypted with a very fast 32-Bit cipher. Sufficient for many purposes but not 100% secure.	An alphanumeric password up to 40 characters, e.g. 3Huv	All table levels. All versions of TurboDB Win. TurboDB Managed 2.x and

			above.
Blowfish	Encryption with the well-known Blowfish algorithm using a 128 bit key.	An alphanumeric password up to 40 characters, e.g. 3Huv	Table level 4 and above. TurboDB Win 5 and above. Not yet supported in TurboDB Managed.
Rijndael	Encryption with the well-known Rijndael algorithm using a 128 bit key. Also known as Advanced Encryption Standard (AES).	An alphanumeric password up to 40 characters, e.g. 3Huv	Table level 4 and above. TurboDB Win 5 and above. TurboDB Managed 2.x and above.
AES	Same as Rijndael.	Same as Rijndael	Same as Rijndael

1.3.3 TurboPL Guide

TurboDB Engine has a set of native built-in functions. These functions have been used for check constraints, calculated indexes and default values in table levels before level 5. Since then all of this functionality is available through TurboSQL, but TurboPL can still be used. When you want to use a TurboPL expression as the formula of a calculated field or an calculated index, you must precede it by the @ sign.

For example: *@RightStr(Column1)* uses the TurboPL interpreter instead of the TurboSQL interpreter to evaluate the formula. This is important for backward compatibility but is not recommended for new or updated tables.

- TurboPL Operators and functions
- TurboPL Search-Conditions

See also

TurboSQL Guide

1.3.3.1 Operators and Functions

- Arithmetic Operators and Functions
- String Operators and Functions
- Date and Time Operators and Functions
- Miscellaneous Operators and Functions

1.3.3.1.1 TurboPL Arithmetic Operators and Functions

These arithmetic operators and functions can be used in TurboPL expressions. They are no more recommended for TurboSQL.

Operators

+	Addition
-	Subtraction
*	Multiplication
/	Real division
div	Integer division

mod	Remainder

Comparisons

<	less
<=	less or equal
=	equal
>=	greater or equal
>	greater
less	less
is	equal
greater	greater
<>	not equal
from...upto	range test
in [...]	set test

Functions

Abs(X: Real): Real	Returns the absolute value of the argument X.
ArcTan(X: Real): Real	Returns the arctangent of a given X.
Cos(X: Real): Real	Returns the cosine of the angle X, in radians.
Exp(X: Real): Real	Returns the value of e raised to the power of X, where e is the base of the natural logarithm.
Frac(X: Real): Real	Returns the fractional part of the argument X.
Int(X: Real): Integer	Returns the integer part of X, that is, X rounded toward zero.
Log(X: Real): Real	Returns the natural log of a real expression.
Round(X: Real; [Scale: Integer]): Real	Returns the value of X rounded to the nearest number with the given scale. Scale can be negative as well.
Sin(X: Real): Real	Returns the sine of the angle in radians.
Sqrt(X: Real): Real	Returns the square root of X.

Compatibility Information

TurboPL is supported only for backward compatibility in tables up to level 4.

1.3.3.1.2 TurboPL String Operators and Functions

These string operators and functions can be used in TurboPL expressions. They are no more recommended for TurboSQL.

Operators

+	concatenation, e.g. EMPLOYEES.FirstName + ' ' + EMPLOYEES.LastName
[]	access to single character, e.g. EMPLOYEES.FirstName[1] + '. ' + EMPLOYEES.LastName

Comparisons

<	less

<=	less or equal
=	equal
>=	greater or equal
>	greater
less	less
equal	equal
greater	greater
<>	not equal
has	case sensitive search of string within another, e.g. 'John Smith' has 'Sm'
like	case insensitive correspondence to mask containing jokers, e.g. 'Smith' like 'sMIth', 'Smith' like 'Sm*', 'Smith' like 'Smit?' (all true)
in [..]	tests, whether an element is contained within the set

Functions

Arguments in brackets are optional.

Asc(C: String): Integer	Returns the ordinal value of the first character in the string.
Chr(N: Integer): String	Returns the character for a specified Unicode value.
Exchange(Source, From, To: String): String	Replaces all occurrences of From in Source by To and returns the modified string.
FillStr(Source, Filler: String; Len: Integer): String	Fills the Source with the Filler up to the given Length and returns the result.
LeftStr(Source: String; Len: Integer): String	Return the left substring of source with given length.
Length(Source: String)	Return the count of characters in Source.
Lower(Source: String): String	Returns the string in lowercase.
LTrim(Source: String): String	Returns Source without any leading white-space.
MemoStr(Memo: MemoField [; Len: Integer]): String	Returns the first Len (default is 255, -1 means all) characters of the content of the memo field in the current record.
NTimes(Source: String; Count: Integer): String	Returns a string that repeats Source Count times.
RealVal(Str: String): Real	Calculates the numeric value of a string expression.
Pos(SubStr, Source: String): Integer	Returns the position of SubStr in Source or 0, if SubStr is not contained in Source.
RightStr(Source: String; Len: Integer)	Returns the Len last characters of Source.
RTrim(Source: String): String	Returns Source without any trailing white-spaces.
Scan(SubStr, Source: String): Integer	Returns the number of occurrences of SubStr in Str.
Str(Num: Real[; Width, Scale: Integer]): String	Returns the alphanumeric representation of Num with given Width and Scale. Width=1 means as needed. The alphanumeric representation of an enumeration value (see column data types) is the name of the value.
Upper(Source: String): String	Returns the string in uppercase.
NewGuid: String	Returns a string denoting a new Globally Unique Identifier.

Compatibility Information

TurboPL is supported only for backward compatibility in tables up to level 4.

1.3.3.1.3 TurboPL Date and Time Operators and Functions

These date and time operators and functions can be used in TurboPL expressions. They are no more recommended for TurboSQL.

Comparison

All numeric comparison operators can be used for time, date and datetime values as well. E.g. Date1 > Date2 if and only if Date1 designates a point in time later then Date2.

Calculations

You can add time spans to dates, subtract time spans from dates and subtract dates from each other to get the time span. The time span is a real number, which indicates the number of days (including a fractional part for the time of day) when calculating with dates and datetimes or the number of minutes (including a fractional part for the seconds and milliseconds) when calculating with times.

If Time1 and Time2 are time values, Date1 and Date2 date values, DateTime1 and DateTime2 datetime variables and TimeSpan1, TimeSpan2 real variables, then the following expressions are meaningful:

```
Time2 - Time1
Time2 - TimeSpan1
Time1 + TimeSpan2
Date2 - Date1
Date2 - TimeSpan1
Date2 + TimeSpan2
DateTime2 - DateTime1
DateTime2 - TimeSpan1
DateTime2 - TimeSpan2
```

Beyond the numeric operators and functions there are also special date and time functions:

CombineDateTime

CombineDateTime(ADate: Date; ATime: Time): DateTime

Puts a date and a time together to form a datetime.

DateStr

DateStr(ADateTime: DateTime): String

Converts a date or a time stamp into a string according to the current date format.

DateTimeStr

DateTimeStr(ADateTime: DateTime; TimePrecision: Integer): String

Converts a time stamp into a string according to the current date and time format. *TimePrecision* indicates the number of time parts to create (minute = 2, second = 3, millisecond = 4)

Day

Day(ADate: DateTime): Integer

Extracts the day out of a date.

DayOfWeek

DayOfWeek(ADateTime: DateTime): String

Returns the name of the day of the week in the current locale (e.g. Wednesday)

Hour

Hour(ADate: DateTime): Integer

Extracts the hour from a time or datetime.

Millisecond

Millisecond(ADate: DateTime): Integer

Extracts the millisecond from a time or datetime.

Minute

Minute(ADate: DateTime): Integer

Extracts the minute from a time or datetime.

Month

Month(ADate: DateTime): Integer

Extracts the month out of a date.

Now

Now: Time

Returns the current time.

Second

Second(ADate: DateTime): Integer

Extracts the second from a time or datetime.

TimeStr

TimeStr(ATime: Time): String

Converts a time or time stamp into a string according to the current time format.

Today

Today: Date

Returns the current date

Week

Week(ADate: DateTime): Integer

Returns the number of the calendar week within the year.

WeekDayNo

WeekDayNo(ADateTime: DateTime): Integer

Returns the day of week as a number between 1 (Monday) and 7 (Sunday)

Year

Year(ADate: DateTime): Integer

Extracts the year out of a date.

Compatibility Information

TurboPL is supported only for backward compatibility in tables up to level 4.

1.3.3.1.4 TurboPL Miscellaneous Operators and Functions

These miscellaneous operators and functions can be used in TurboPL expressions. They are no more recommended for TurboSQL.

HexStr(Value: Integer [; Digits: Integer])	Returns the hexadecimal representation of a number with at least *Digits* digits.
CurrentRecordId(TableName)	Returns the last used record id of the given table. Using this function, it is possible to enter linked records in multiple tables within one compound statement.

Compatibility Information

TurboPL is supported only for backward compatibility in tables up to level 4.

1.3.3.2 Search-Conditions

- Filter search-conditions
- Full-text search conditions

1.3.3.2.1 Filter Search-Conditions

Filter search-conditions are used, when entering check constraints for a TurboDB table via TurboDB Viewer or via the VCL *TTdbTable* component. They are very similar to TurboSQL search-conditions with a few exceptions:

- Date, time and number formats are based on the local settings
- * and ? are allowed as jokers as well as % and _
- Sets are written in brackets instead of parenthesis

Examples

(*Name*, *Amount*, *Amount1*, *Amount2* and *Date-of-birth* are table columns.)

```
Name = 'Smith'
Name like 'Smi*'  (deprecated, prefer % in place of *)
Name like 'Smi%'
Name like 'Smit?'  (deprecated, prefer _ in place of ?)
Name like 'Smit_'
Name has 'mit'
LeftStr(Name, 2) = 'Sm'
Length(Name) > 4
Amount = 13546.45
Amount < 13546.46
Amount < 345.67 or Amount > 567.89 (note that the decimal point depends on your local settings)
Amount1 * 0.3 > Amount2 * 0.8
Amount is not null (includes all records that have a value for Amount)
Date-of-birth = '4/20/1962' (note that the date format may depend on your local settings)
Date-of-birth < '4/20/1962'
Date-of-birth between '4/1/1962' and '4/30/1962'
Year(Date-of-birth) = 1962
```

```
Date-of-birth is null  (includes all records that have no value for
Date-of-birth)
```

Rules for Quoting and Escaping

Single quotes are used for string literals. If you need a quote within a string, which is terminated with the same kind of quote, duplicate the quote:

```
'My "quote"' -> My "quote"
```

```
'My ''quote''' -> My 'quote'
```

If a table column has a name, which is also a keyword, you can use double quotes or brackets to specify the meaning:

```
Length("Password") > 8
```

```
Week([Date]) = 18
```

TurboDB offers powerful functions and operators for use in search-conditions, e.g. *like*, *between...and...*, *LeftStr*, *Year* and many others. Refer to Operators and Functions for a complete reference. Comparisons can be combined using the logical operators *and*, *or* and *not*.

Compatibility Information

TurboPL is supported only for backward compatibility in tables up to level 4.

1.3.3.2.2 Full-text Search-Conditions

Full-text search-conditions are used with the TurboSQL contains predicate and the VCL *TTdbTable.WordFilter* property. A full-text search-condition is basically a list of keywords, separated by "+", "," or "-". These characters mean:

, or space	both keywords must occur in the record
+ or /	one of the keywords must occur in the record
-	the keyword must not occur in the record

The alternate character (space and slash) are only available as of table level 4. The keyword itself can contain the jokers "?" and "*" to represent any single character or any substring respectively.

Examples

Database	Finds *Database*, *database*, *dataBase*, ...
Database*	Finds *database*, *Databases*, *DatabaseDriver*, ...
Data?ase	Finds *Database*, *dataCase*, ...
Database, Driver	Record must contain the words *Database* and *Driver*
Database Driver	Same as above for table level 4
Database, Driver, ODBC	Record must contain the words *Database*, *Driver* and *ODBC*
Database Driver ODBC	Same as above for table level 4
Database + Driver	Record must contain either the word *Database* or the word *Driver* or both
Database/Driver	Same as above for table level 4
Database + Driver + ODBC	Record must contain either the word *Database* or the word *Driver* or the word *ODBC*
Database/Driver/ODBC	Same as above for table level 4
Database Driver ODBC/OLE	Record must contain the word *Database* and the word *Driver* and either the word *ODBC* or the word *OLE*
-Database	Record must not contain the word *Database*
Database - Driver	Record must contain the word *Database* but not the word *Driver*

Compatibility Information

TurboPL is supported only for backward compatibility in tables up to level 4.

1.3.4 TurboSQL Guide

TurboSQL is a subset of SQL 92 that contains all of minimal SQL as described by the MS ODBC specification and is very similar to Local SQL used with Embarcadero Database Engine.

Conventions

- Table Names
- Column Names
- String Literals
- Date Formats
- Time Formats
- DateTime Formats
- Boolean Literals
- Table Correlation Names
- Column Correlation Names
- Command Parameters
- Embedded Comments

Data Manipulation Language

- Overview
- DELETE Statement
- FROM Clause
- GROUP BY Clause
- INSERT Statement
- ORDER BY Clause
- SELECT Statement
- UPDATE Statement
- WHERE Clause

Data Definition Language

- Overview
- CREATE TABLE Command
- ALTER TABLE Command
- CREATE INDEX Command
- DROP Command
- Column Data Types

Programming Language

- Overview
- CALL Statement

- CREATE FUNCTION Statement
- CREATE PROCEDURE Statement
- CREATE AGGREGATE Statement
- DROP FUNCTION/PROCEDURE/AGGREGATE Statement
- DECLARE Statement
- IF Statement
- SET Statement
- WHILE Statement
- Exchanging Parameters with .NET Assemblies

1.3.4.1 TurboSQL vs. Local SQL

TurboSQL distinguishes itself in some aspects from Embarcadero's Local SQL:

- In TurboSQL you can enter date, time and datetime literals without quotes, the format is dd.mm.yyyy and HH:mm and dd.mm.yyyy_HH:mm:ss.ms
- TurboDB allows you to issue multiple commands within one statement separated by semicolon.
- TurboDB can rename and modify existing table columns in the ALTER TABLE command.

1.3.4.2 Conventions

1.3.4.2.1 Table Names

Like the ANSI standard TurboSQL confines each table name to a single word comprised of alphanumeric characters and the underscore symbol "_"

```
SELECT *
FROM customer
```

TurboSQL supports full file and path specifications in table references. Table references with path or filename extensions must be enclosed in double quotation marks. For example:

```
SELECT *
FROM "parts.dat"
```

```
SELECT *
FROM "c:\sample\parts.dat"
```

If you omit the file extension for a local table name, ".dat" is assumed.

1.3.4.2.2 Column Names

Like the ANSI-standard TurboSQL confines each column name to a single word comprised of alphanumeric characters and the underscore symbol "_". To distinguish similar column names from different tables preface the table name.

```
SELECT Employee_Id
FROM Employee
```

or

```
SELECT Employee.Employee_Id
FROM Employee
```

In addition, TurboSQL can use the German umlauts for column names and table names:

```
SELECT Kürzung
```

```
FROM Beiträge
```

For using column names that contain spaces or other special characters and for distinguishing column names from e.g. function names, you can enclose the column name in brackets or double quotes:

```
SELECT "Employee Id", [Employee Id]
FROM [Update]
```

1.3.4.2.3 String Literals

String literals are enclosed in single quotes. To denote a single quote within a string literal, insert two of them. These are samples for valid string literals:

```
'This is a string literal'
```

```
'This is a ''single-quoted'' string literal'
```

```
'This is a "double-quoted" string literal'
```

And these are invalid:

```
'This isn't a valid string literal'
```

Note

Earlier version of TurboDB allowed also double quotes for string literals. For enhanced SQL compatibility, double quotes are now restricted to denote table and column identifiers.

1.3.4.2.4 Date Formats

Date values can be indicated either in the TurboDB proprietary format which does not require quotation marks (dd.mm.yyyy) or in three different standard date formats. Where local date formats are allowed, only the TurboDB format and the current local format are valid. In these situations the three standard date formats are not accepted.

The native format is dd.mm.yyyy. This format is a very logical one and can not be mistaken by the parser for arithmetic calculations. For this reason, it is not necessary to enclose such a date literal in quotation marks. Example:

```
SELECT * FROM orders
WHERE saledate <= 31.12.2001
```

searches for sales on 31 December 2001. This format is always valid and always interpreted in the same way. You should prefer it wherever you do not want the date format to adjust to the local settings on the computer.

The quoted date formats are valid wherever local date formats are not allowed, for example in all SQL statements. There is an American, an International and a European date format. The quoted string is preceded by the keyword *DATE*:

```
SELECT * FROM orders
WHERE saledate <= DATE'12/31/2001'
```

or

```
SELECT * FROM orders
WHERE saledate <= DATE'2001-12-31'
```

or

```
SELECT * FROM orders
WHERE saledate <= DATE'31.12.2001'
```

Leading zeros for the month and day fields are optional. If the century is not specified for the year, TurboDB assumes the 20th century for years from 50 to 99 and the 21th century for years from 00 to 49.

You can omit the keyword *DATE* where the type of the string is obvious like in the above examples.

Example

```
SELECT * FROM orders
WHERE (saledate > 1.1.89) AND (saledate <= 31.12.20)
```

searches for sales between the January 1st 1989 and the December 31 2020.

1.3.4.2.5 Time Formats

Time values can be indicated either in the TurboDB proprietary format which does not require quotation marks (hh:mm:ss) or in two different standard time formats. Where local date formats are allowed, only the TurboDB format and the current local format are valid. In these situations the quoted standard time formats are not accepted.

The native format expects time literals to be in the format HH:mm:ss.ttt; where HH are the hours and mm the minutes. TurboSQL uses the 24 hour scale, that is 2:10 is in the early morning (2:10 AM) while 14:10 is in the early afternoon (2:10 PM). This time literal must not be enclosed in quotation marks.

```
INSERT INTO WorkOrder
(ID, StartTime) VALUES ('B00120', 22:30)
```

This format is always valid and always interpreted in the same way. You should prefer it wherever you do not want the time format to adjust to the local settings on the computer.

If you prefer, you can enter the time value in the American format *hh:mm:ss am/pm*. In order to do this, you must enclose the time literal in single quotes and proceed it by the keyword *TIME*:

```
INSERT INTO WorkOrder
(ID, StartTime) VALUES ('B00120', TIME'10:30:00 pm')
```

Where the type of the string is obvious like in the above example, you can omit the keyword *TIME*. An example where you can not omit it is this one:

```
SELECT StartTime - TIME'12:00:00 pm' FROM WorkOrder
```

Note
When you want to use the native format without enclosing quotes with the Delphi/C++ Builder component *TTdbQuery*, it will create a parameter because the VCL parser for SQL commands recognizes the colon as the starting character of a parameter. You can either delete it or ignore it, the statement will be executed correctly anyway.

1.3.4.2.6 Timestamp Formats

Timestamp values can be indicated either in the TurboDB proprietary format which does not require quotation marks (*dd.mm.yyyy_hh:mm:ss*) or in three different standard timestamp formats. Where local date formats are allowed, only the TurboDB format and the current local format are valid. In these situations the quoted standard timestamp formats are not accepted.

The native format for timestamp literals is composed of a date literal and a time literal separated by an underscore '_'. This format is used without quotes:

```
SELECT * FROM WorkOrder
WHERE StartTime >= 31.1.2001_14:10:00.500
```

This format is always valid and always interpreted in the same way. You should prefer it wherever you do not want the timestamp format to adjust to the local settings on the computer.

The other way to specify a timestamp is to proceed it by the keyword TIMESTAMP and enclose it in single quotes. Again, we have three different representations here:

The American timestamp format:

```
SELECT * FROM WorkOrder
WHERE StartTime >= TIMESTAMP'1/31/2001 2:10:00 pm'
```

The international timestamp format:

```
SELECT * FROM WorkOrder
```

```
WHERE StartTime >= TIMESTAMP'2001-1-31 14:10:00'
```

The European timestamp format:

```
SELECT * FROM WorkOrder
WHERE StartTime >= TIMESTAMP'31.1.2001 14:10:00'
```

When the nature of the string is obvious like in the above samples, you can omit the keyword *TIMESTAMP*.

Note
When you want to use the native format without enclosing quotes with the Delphi/C++ Builder component *TTdbQuery*, it will create a parameter because the VCL parser for SQL commands recognizes the colon as the starting character of a parameter. You can either delete it or ignore it, the statement will be executed correctly anyway.

1.3.4.2.7 Boolean Literals

The Boolean literal values *True* and *False* can be written with or without single quotes. Uppercase and lowercase is ignored.

```
SELECT *
FROM transfers
WHERE (paid = 'True') AND NOT (incomplete = FALSE)
```

1.3.4.2.8 Table Correlation Names

Table correlation names are used to explicitly associate a column with the table from which it comes. This is especially useful when multiple columns of the same name appear in the same query, typically in multi-table queries. A table correlation name is defined by following the table reference in the FROM clause of a SELECT query with a unique identifier. This identifier, or table correlation name, can then be used to prefix a column name.

If the table name is not a quoted string, the table name is the default implicit correlation name. An explicit correlation name the same as the table name need not be specified in the FROM clause and the table name can prefix column names in other parts of the statement.

```
SELECT *
FROM "/home/data/transfers.dat" transfers
WHERE transfers.incomplete = False
```

1.3.4.2.9 Column Correlation Names

Use the optional keyword AS to assign a correlation name to a column, aggregated value, or literal. In the statement below, the tokens Sub and Word are column correlation names.

```
SELECT SUBSTRING(company FROM 1 FOR 1) AS sub, Text word
FROM customer
```

1.3.4.2.10 Command Parameters

TurboSQL uses named statement parameters. They are preceded by a colon:

```
INSERT INTO Customers (Name) VALUES(:Name)
```

The parameter name is the identifier excluding the colon, i.e. *Name* in this case. Whenever you refer to a command parameter in one of the API functions or working with a component library, indicate the identifier without the colon.

When working with the ODBC interface, unnamed parameters are supported as well:

```
INSERT INTO Customers (Name) VALUES (?)
```

1.3.4.2.11 Comments

There are two ways to embed comments into your TurboSQL statement comparable to the comments available in C++. Either you may use /* and */ to enclose the comment or you use // to begin a comment that lasts until the end of the current line.

Example

```
/* This finds all requests that have come in in the period of time we
are looking at. */
SELECT * FROM Request
// Requests from the time before the Euro came
WHERE Date < '1/1/2002'
```

1.3.4.3 System Tables

TurboDB uses system tables to store management information and to provide the information schema to the user. The following tables correspond to their counterparts in SQL 92

It depends on the management level of the database whether these tables are permanent or temporary. In both cases you can query on them.

sys_UserTables	Lists the user tables of the database.
sys_UserColumns	Lists the visible columns of the user tables.
sys_UserTableConstraints	Lists the names of all keys, checks and foreign keys of all user tables.
sys_UserKeyColumns	Lists all columns from user tables that make part of a (primary, candidate or foreign) key.
sys_UserCheckConstraints	Lists the check constraints including the check condition.
sys_UserReferentialConstraints	Indicates the referenced unique constraint and the referential action for each foreign key in the database.
sys_UserIndexes	Lists the indexes of all user tables.
sys_UserIndexColumns	Lists all indexed columns of all user tables.
sys_UserRoutines	Lists all stored procedures of the database.

Examples:

```
select ColumnName, DataType from sys_UserColumns where TableName =
'TableA'
```

Displays the columns and their data types for table *TableA*.

```
select I.TableName, I.IndexName, C.ColumnName
from sys_UserIndexes I join sys_UserIndexColumns C on I.TableName = C.
TableName and I.IndexName = C.IndexName
where I.IsUnique = True
```

Displays the columns of all unique indexes.

1.3.4.4 Data Manipulation Language

TurboSQL includes the following commands, clauses, functions and predicates for the DML:

Statements

DELETE	Deletes one or more rows from a table.
INSERT	Inserts one or more rows into a table.
SELECT	Retrieves data from tables.
UPDATE	Modifies one or more existing rows in a table.

Clauses

FROM	Specifies the tables from which a SELECT statement retrieves data.
GROUP BY	Combines rows with column values in common into single rows.
HAVING	Specifies filtering conditions for a SELECT statement.
ORDER BY	Sorts the rows retrieved by a SELECT statement.
WHERE	Specifies filtering conditions for a SELECT, UPDATE or DELETE statement.
Sub-Queries	Comparisons to the result of a different query with IN, ANY, SOME, ALL and EXISTS.

Functions, Predicates and Operators

General Functions and Operators	Calculations and comparisons with numbers, strings, timestamps etc.
Arithmetic Functions and Operators	Calculations with numbers
String Functions and Operators	Calculations with strings
Date and Time Functions and Operators	Calculations with date and time
Aggregation Functions	Statistics
Miscellaneous Functions and Operators	Calculations that do not fit in one of the other categories
Table operators	Combine two table into a new one
Sub-Queries	Compare rows with the result of another query
Full-Text Search	Searching for arbitrary keywords anywhere in a row.

1.3.4.4.1 DELETE Statement

Deletes one or more rows from a table.

```
DELETE FROM table_reference
[WHERE predicates]
```

Description

Use DELETE to delete one or more rows from an existing table.

```
DELETE FROM [employee]
```

The optional WHERE clause restricts row deletions to a subset of rows in the table. If no WHERE clause is specified, all rows in the table are deleted.

```
DELETE FROM [employee]
WHERE empno > 2300
```

The table reference cannot be passed to the DELETE statement via a parameter.

Important Note:

The DELETE statement without WHERE clause deletes all rows of a table without checking constraints like foreign keys. This is a feature to provide fast table emptying.

1.3.4.4.2 FROM Clause

Specifies the tables from which a SELECT statement retrieves data.

```
FROM table_reference [, table_reference...]
```

Description

Use a FROM clause to specify the table or tables from which a SELECT statement retrieves data. The value for a FROM clause is a comma-separated list of table names. Specified table names must follow TurboSQL naming conventions for tables. The following examples show different ways, how the from clause can look like:

```
SELECT * FROM [customer]
```

```
SELECT * FROM
customer, orders
```

```
SELECT * FROM
customer JOIN orders ON orders.CustNo = customer.CustNo
```

Applicability

SELECT

1.3.4.4.3 GROUP BY Clause

Combines rows with column values in common into single rows.

```
GROUP BY column_reference [, column reference...]
```

Description

Use a GROUP BY clause to combine rows with the same column values into a single row. The criteria for combining rows is based on the values in the columns specified in the GROUP BY clause. The purpose for using a GROUP BY clause is to combine one or more column values (aggregate) into a single value and provide one or more columns to uniquely identify the aggregated values. A GROUP BY clause can only be used when one or more columns have an aggregate function applied to them.

The value for the GROUP BY clause is a comma-separated list of columns. Each column in this list must meet the following criteria:

- Be in one of the tables specified in the FROM clause of the query.
- Be in the SELECT clause of the query.
- Cannot have an aggregate function applied to it.

When a GROUP BY clause is used, all table columns in the SELECT clause of the query must meet at least one of the following criteria, or it cannot be included in the SELECT clause:

- Be in the GROUP BY clause of the query.
- Be in the subject of an aggregate function.

Literal values in the SELECT clause are not subject to the preceding criteria.

The distinctness of rows is based on the columns in the column list specified. All rows with the same values in these columns are combined into a single row (or logical group). Columns that are the subject of an aggregate function have their values across all rows in the group combined. All columns not the subject of an aggregate function retain their value and serve to distinctly identify the group. For example, in the SELECT statement below, the values in the SALES column are aggregated (totaled) into groups based on distinct values in the COMPANY column. This produces total sales for each company.

```
SELECT company, SUM(sales) AS TOTALSALES
FROM sales1998
GROUP BY company
ORDER BY company
```

A column may be referenced in a GROUP BY clause by a column correlation name, instead of actual column names. The statement below forms groups using the first column, COMPANY, represented by the column correlation name Co.

```
SELECT company AS Co, SUM(sales) AS TOTALSALES
FROM sales1998
GROUP BY Co
ORDER BY 1
```

Notes

- Derived values (calculated fields) cannot be used as the basis for a GROUP BY clause.
- Column references cannot be passed to an GROUP BY clause via parameters.

Applicability

SELECT, when aggregate functions used

1.3.4.4.4 HAVING Clause

Specifies filtering conditions for a SELECT statement.

```
HAVING predicates
```

Description

Use a HAVING clause to limit the rows retrieved by a SELECT statement to a subset of rows where aggregated column values meet the specified criteria. A HAVING clause can only be used in a SELECT statement when:

- The statement also has a GROUP BY clause.
- One or more columns are the subjects of aggregate functions.

The value for a HAVING clause is one or more logical expressions, or predicates, that evaluate to true or false for each aggregate row retrieved from the table. Only those rows where the predicates evaluate to true are retrieved by a SELECT statement. For example, the SELECT statement below retrieves all rows where the total sales for individual total sales exceed $1,000.

```
SELECT company, SUM(sales) AS TOTALSALES
FROM sales1998
GROUP BY company
HAVING (SUM(sales) >= 1000)
ORDER BY company
```

Multiple predicates must be separated by one of the logical operators OR or AND. Each predicate can be negated with the NOT operator. Parentheses can be used to isolate logical comparisons and groups of comparisons to produce different row evaluation criteria.

A SELECT statement can include both a WHERE clause and a HAVING clause. The WHERE clause filters the data to be aggregated, using columns not the subject of aggregate functions. The HAVING clause then further filters the data after the aggregation, using columns that are the subject of aggregate functions. The SELECT query below performs the same operation as that above, but data limited to those rows where the STATE column is "CA".

```
SELECT company, SUM(sales) AS TOTALSALES
FROM sales1998
WHERE (state = 'CA')
GROUP BY company
HAVING (SUM(sales) >= 1000)
ORDER BY company
```

Note

A HAVING clause filters data after the aggregation of a GROUP BY clause. For filtering based on row values prior to aggregation, use a WHERE clause.

Applicability

SELECT with GROUP BY

1.3.4.4.5 INSERT Statement

Adds one or more new rows of data in a table

```
INSERT INTO table_reference
[(columns_list)]
VALUES (update_atoms)
```

Description

Use the INSERT statement to add new rows of data to a table.

Use a table reference in the INTO clause to specify the table to receive the incoming data.

The columns list is a comma-separated list, enclosed in parentheses, of columns in the table and is optional. The VALUES clause is a comma-separated list of update atoms, enclosed in parentheses. If no columns list is specified, incoming update values (update atoms) are stored in fields as they are defined sequentially in the table structure. Update atoms are applied to columns in the order the update atoms are listed in the VALUES clause. There must also be as many update atoms as there are columns in the table.

```
INSERT INTO [holdings]
VALUES (4094095, 'BORL', 5000, 10.500, 2.1.1998)
```

If an explicit columns list is stated, incoming update atoms (in the order they appear in the VALUES clause) are stored in the listed columns (in the order they appear in the columns list). NULL values are stored in any columns that are not in a columns list.

```
INSERT INTO [customer]
(custno, company)
VALUES (9842, 'dataweb GmbH')
```

To add rows to one table from another, omit the VALUES keyword and use a subquery as the source for the new rows.

```
INSERT INTO [customer]
(custno, company)
SELECT custno, company
FROM [oldcustomer]
```

1.3.4.4.6 ORDER BY Clause

Sorts the rows retrieved by a SELECT statement.

```
ORDER BY column_reference [, column_reference...] [ASC|DESC]
```

Description

Use an ORDER BY clause to sort the rows retrieved by a SELECT statement based on the values from one or more columns.

The value for the ORDER BY clause is a comma-separated list of column names. The columns in this list must also be in the SELECT clause of the query statement. Columns in the ORDER BY list can be from one or multiple tables. A number representing the relative position of a column in the SELECT clause may be used in place of a column name. Column correlation names can also be used in an ORDER BY clause columns list.

Use *ASC* (or *ASCENDING*) to force the sort to be in ascending order (smallest to largest), or *DESC* (or *DESCENDING*) for a descending sort order (largest to smallest). When not specified, *ASC* is the implied by default.

The statement below sorts the result set ascending by the year extracted from the *lastinvoicedate* column, then descending by the *state* column, and then ascending by the uppercase conversion of the *company* column.

```
SELECT EXTRACT(YEAR FROM lastinvoicedate) AS YY, state, UPPER(company)
FROM customer
ORDER BY YY DESC, state ASC, 3
```

Column references cannot be passed to an ORDER BY clause via parameters.

Applicability

SELECT

1.3.4.4.7 SELECT Statement

Retrieves data from tables.

```
SELECT [TOP number] [DISTINCT] * | column_list
FROM table_reference
[WHERE predicates]
[ORDER BY order_list]
[GROUP BY group_list]
[HAVING having_condition]
```

Description

Use the SELECT statement to

- Retrieve a single row, or part of a row, from a table, referred to as a singleton select.
- Retrieve multiple rows, or parts of rows, from a table.
- Retrieve related rows, or parts of rows, from a join of two or more tables.

The SELECT clause defines the list of items returned by the SELECT statement. The SELECT clause uses a comma-separated list composed of: table columns, literal values, and column or literal values modified by functions. Literal values in the columns list may be passed to the SELECT statement via parameters. You cannot use parameters to represent column names. Use an asterisk to retrieve values from all columns.

Columns in the column list for the SELECT clause may come from more than one table, but can only come from those tables listed in the FROM clause. See Relational Operators for more information on using the SELECT statement to retrieve data from multiple tables. The FROM clause identifies the table(s) from which data is retrieved.

If TOP is specified in the statement, the number of rows in the subset is limited to the given number. Top is evaluated after all other clauses and therefore refers to the sorted or grouped result set in case the order by clause and/or the group by clause are present.

If the DISTINCT keyword is present, duplicate rows in the result table are suppressed. DISTINCT cannot be used together with GROUP BY. If a SELECT statement contains both GROUP BY and DISTINCT, the DISTINCT keyword is ignored.

The following statement retrieves data for two columns in all rows of a table.

```
SELECT custno, company
FROM orders
```

See also

JOIN, UNION, INTERSECT, EXCEPT

1.3.4.4.8 UPDATE Statement

Modifies one or more existing rows in a table.

```
UPDATE table_reference
SET column_ref = update_value [, column_ref = update_value...]
[WHERE predicates]
```

Description

Use the UPDATE statement to modify one or more column values in one or more existing rows in a table.

Use a table reference in the UPDATE clause to specify the table to receive the data changes.

The SET clause is a comma-separated list of update expressions. Each expression is composed of the name of a column, the assignment operator (=), and the update value for that column.

```
UPDATE salesinfo
SET taxrate = 0.0825
WHERE (state = 'CA')
```

The optional WHERE clause restricts updates to a subset of rows in the table. If no WHERE clause is specified, all rows in the table are updated using the SET clause update expressions.

By using a subquery the values can also be taken from a different table:

```
UPDATE salesinfo
SET taxrate = (SELECT newesttaxrate FROM [globals])
```

See also

INSERT, DELETE

1.3.4.4.9 WHERE Clause

Specifies filtering conditions for a SELECT or UPDATE statement.

```
WHERE predicates
```

Description

Use a WHERE clause to limit the effect of a SELECT or UPDATE statement to a subset of rows in the table. Use of a WHERE clause is optional.

The value for a WHERE clause is one or more logical expressions, or predicates, that evaluate to TRUE or FALSE for each row in the table. Only those rows where the predicates evaluate to TRUE are retrieved by a SELECT statement or modified by an UPDATE statement. For example, the SELECT statement below retrieves all rows where the STATE column contains a value of 'CA'.

```
SELECT company, state
FROM customer
WHERE state = 'CA'
```

Multiple predicates must be separated by one of the logical operators OR or AND. Each predicate can be negated with the NOT operator. Parentheses can be used to isolate logical comparisons and groups of comparisons to produce different row evaluation criteria. For example, the SELECT statement below retrieves all rows where the STATE column contains a value of "CA" and those with a value of "HI".

```
SELECT company, state
FROM customer
WHERE (state = 'CA') OR (state = 'HI')
```

The SELECT statement below retrieves all rows where the SHAPE column is *round* or *square*, but only if the COLOR column also contains *red*. It would not retrieve rows where, for example, the SHAPE is *round* and the COLOR *blue*.

```
SELECT shape, color, cost
FROM objects
WHERE ((shape = 'round') OR (shape = 'square')) AND (color = 'red')
```

But without the parentheses to override the order of precedence of the logical operators, as in the statement that follows, the results are very different. This statement retrieves the rows where the SHAPE is round, regardless of the value in the COLOR column. It also retrieves rows where the SHAPE column is *square,* but only when the COLOR column contains *red*. Unlike the preceding variation of this statement, this one would retrieve rows where the SHAPE is *round* and the COLOR *blue*.

```
SELECT shape, color, cost
FROM objects
WHERE shape = 'round' OR shape = 'square' AND color = 'red'
```

Note

A WHERE clause filters data prior to the aggregation of a GROUP BY clause. For filtering based on aggregated values, use a HAVING clause.

Applicability
SELECT, UPDATE, DELETE

1.3.4.4.10 General Functions and Operators

There is a list of functions and operators that can be used within TurboSQL expressions. This list is composed of a few standard SQL functions and a lot more additional TurboDB functions.

=

Syntax
```
expr1 = expr2
```
Description
Tests for equality.

<

Syntax
```
expr1 < expr2
```
Description
Tests whether expression *expr1* is lower than *expr2*.

<=

Syntax
```
expr1 <= expr2
```
Description
Tests whether expression *expr1* is lower or equal than *expr2*.

>

Syntax
```
expr1 > expr2
```
Description
Tests whether expression *expr1* is greater than *expr2*.

>=

Syntax
```
expr1 >= expr2
```
Description
Tests whether expression *expr1* is greater or equal than *expr2*.

BETWEEN ... AND ...

Syntax
```
expr1 BETWEEN expr2 AND expr3
```
Description
Tests whether expression *expr1* is greater or equal than *expr2* and lower or equal than *expr3*.

IN

Syntax
```
expr IN (expr1, expr2, expr3, ...)
```
Description

Tests whether *expr* is equal to one of the expressions *expr1*, *expr2*, *expr3*, ...

AND

Syntax

```
cond1 AND cond2
```

Description

Tests whether both *cond1* and *cond2* are true.

OR

Syntax

```
cond1 OR cond2
```

Description

Tests whether at least one of *cond1* and *cond2* is true.

NOT

Syntax

```
NOT cond
```

Description

Tests whether *cond* is false.

CASE

Syntax

```
CASE
  WHEN cond1 THEN expr1
  WHEN cond2 THEN expr2
  ...
  [ELSE exprN]
END
```

```
CASE expr
  WHEN exprA1 THEN exprB1
  WHEN exprA2 THEN exprB2
  ...
  [ELSE exprBN]
END
```

Description

The first form of the case operation determines the first expression for which the condition is true. The second one returns the B expression, who's A expression is equal to *expr*.

Samples

```
CASE WHEN Age < 8 THEN 'infant'WHEN Age < 18 THEN 'teenager' WHEN Age <
30 THEN 'twen' ELSE 'adult' END
CASE Status WHEN 0 THEN 'OK' WHEN 1 THEN 'WARNING' WHEN 2 THEN 'ERROR'
END
```

CAST

Syntax

```
CAST(value AS type [COLLATE collation])
```

Description

Converts the value to the given type if possible. The cast operation may cut off strings and loose precision of decimal numbers. If the conversion is not possible, CAST raises an error.
Casting to string types optionally allows to set a custom sort collation.

Examples

```
CAST(time AS CHAR(10)) --Converts the time in its string representation
CAST(time AS CHAR(3)) --Displays only the first three characters
CAST(username AS CHAR(50) Collate German_cs_as) --Sets a custom sort
collation on field username
CAST(amount AS INTEGER)) --Looses the digits after the decimal point
CAST('abc' AS BIGINT) --Raises a conversion error
CAST(34515 AS BYTE) --Raises an overflow error
```

See also

General Functions and Operators
Arithmetic Functions and Operators
String Functions and Operators
Date and Time Functions and Operators
Aggregation Functions
Miscellaneous Functions and Operators

1.3.4.4.11 Arithmetic Functions and Operators

This is a list of arithmetic functions and operators that can be used in TurboSQL.

+

Syntax

```
value1 + value2
```

Description

Calculates the sum of two numbers.

-

Syntax

```
value1 - value2
```

Description

Calculates the difference of two numbers.

*****

Syntax

```
value1 * value2
```

Description

Calculates the product of two numbers.

/

Syntax

```
value1 / value2
```

Description

Calculates the quotient of two numbers.

%

Syntax

```
value1 % value2
```

Description

Calculates the modulo of two integral numbers.

Compatibility Information

This operator is only available in TurboDB Managed.

ARCTAN

Syntax

```
ARCTAN(value)
```

Description

Calculates the arcus tangens of value.

CEILING

Syntax

```
CEILING(value)
```

Description

Calculates the smallest integral number greater than, or equal to, the given value.

Example

```
CEILING(-3.8) --returns -3.0
CEILING(3.8) --returns 4.0
```

COS

Syntax

```
COS(value)
```

Description

Calculates the cosine of value.

DIV

Syntax

```
a div b
```

Description

Integer division

Example

```
35 div 6 --returns 5
-35 div 6 --returns -5
35 div -6 --returns -5
-35 div -6 --returns 5
```

EXP

Syntax

```
EXP(:X DOUBLE) RETURNS DOUBLE
```

Description

Calculates the exponential of value (to base e)

FLOOR

Syntax

```
FLOOR(:X DOUBLE) RETURNS DOUBLE
```

Description

Calculates the largest integral number less than or equal to the given value.

Example

```
FLOOR(-3.8) --returns -4.0
FLOOR(3.8) --returns 3.0
```

FRAC

Syntax

```
FRAC(:X DOUBLE) RETURNS DOUBLE
```

Description

Calculates the fractional part of real number.

Example

```
FRAC(-3.8) --returns -0.8
FRAC(3.8) --returns 0.8
```

INT

Syntax

```
INT(:X DOUBLE) RETURNS BIGINT
```

Description

Calculates the integral part of a real number as an integer number.

Example

```
INT(-3.8) --returns -3
INT(3.8) --returns 3
```

LOG

Syntax

```
LOG(:X DOUBLE) RETURNS BIGINT
```

Description

Calculates the natural logarithm of a real number.

MOD

Syntax

```
a mod b
```

Description

Remainder of the integer division. *a mod b = a - (a div b) * b* always holds.

Example

```
35 mod 6 --returns 5
35 mod -6 --returns 5
-35 mod 6 --returns 5
-35 mod -6 --returns 5
```

ROUND

Syntax

```
ROUND(:X DOUBLE [, :Precision BYTE]) RETURNS DOUBLE
```

Description

Rounds the value to the given number of digits.

Compatibility Information

This function is only available in TurboDB Win.

Example

```
ROUND(3.141592, 3) --returns 3.142
```

SIN

Syntax

```
SIN(:X DOUBLE) RETURNS DOUBLE
```

Description

Calculates the sine of a value.

SQRT

Syntax

```
SQRT(:X DOUBLE) RETURNS DOUBLE
```

Description

Calculates the square root of a value.

See also

General Functions and Operators
Arithmetic Functions and Operators
String Functions and Operators
Date and Time Functions and Operators
Aggregation Functions
Miscellaneous Functions and Operators

1.3.4.4.12 String Operators and Functions

This is a list of string operators and functions that can be used in TurboSQL.

||

Syntax

```
string1 || string2
```

Description

Concatenates the two strings.

ASCII

Syntax

```
ASCII(string)
```

Description

Calculates the code point of the first character in the string. Returns NULL if the string is NULL or empty.

CHAR_LENGTH

Syntax

```
CHAR_LENGTH(string)
```

Description

Calculates the number of characters in the string.

HEXSTR

Syntax

```
HEXSTR(number, width)
```

Description

Calculates an hexadecimal representation of number with at least width characters.

Example

```
HEXSTR(15, 3) --returns '00F'
```

LEFTSTR

Syntax

```
LEFTSTR(string, count)
```

Description

Calculates the *count* left characters of *string*.

LEN

Syntax

```
LEN(string)
```

Description

Same as *CHAR_LENGTH*. Prefer *CHAR_LENGTH* as it is standard SQL.

LIKE

Syntax

```
string1 [NOT] LIKE string2 [ESCAPE char1]
```

Description

Compares the two strings as defined in standard SQL using the two joker characters % and _. Define an escape character to use the joker characters as regular characters.

Examples

```
'Woolfe' LIKE 'Woo%'
Name LIKE '_oolfe'
Name LIKE '100^% pure orange juice' ESCAPE '^'
```

LOWER

Syntax

```
LOWER(string)
```

Description

Returns the string in lower case.

RIGHTSTR

Syntax

```
RIGHTSTR(string, count)
```

Description

Calculates the *count* right characters of *string*.

STR

Syntax

```
STR(number, width, scale, thousand_separator, fill_character,
decimal_separator)
STR(enumeration_column_reference)
```

Description

The first variant calculates a string representation of the number with the given formatting.

The second variant calculates the string representation of the enumeration value.

Example

```
STR(3.14159, 10, 4, ',', '*', '.') --returns ****3.1416
```

SUBSTRING

Syntax

```
SUBSTRING(string FROM start [FOR length])
```

Returns a partial string of length *length* from *string* starting at *start*.

TRIM

Syntax

```
TRIM([kind [char] FROM] string)
```

Description

Returns a string without leading or trailing characters.
Kind is one of LEADING, TRAILING, BOTH. The default for kind is BOTH.
Char is the character that is trimmed away. The default for char is the space.

Examples

All these expressions return 'Carl':

```
TRIM('  Carl      ')
TRIM(LEADING FROM '   Carl')
TRIM(TRAILING FROM 'Carl    ')
TRIM(BOTH 'x' FROM 'xxCarlxx')
```

UPPER

Syntax

```
UPPER(string)
```

Description

Returns the string in upper case.

See also

General Functions and Operators
Arithmetic Functions and Operators
String Functions and Operators
Date and Time Functions and Operators
Aggregation Functions
Miscellaneous Functions and Operators

1.3.4.4.13 Date and Time Functions and Operators

This is a list of date and time functions and operators that can be used in TurboSQL.

+

Syntax

```
date + days
timestamp + days
time + minutes
```

Description

Adds a number of days to a date or timestamp. Adds a number of minutes to a time value.

Examples

```
CURRENT_DATE + 1 --Tomorrow's date
CURRENT_TIMESTAMP + 1 --Tomorrow's time exactly like now
CURRENT_TME + 60 --One hour from now
CURRENT_TIME + 0.25 --15 seconds later
```

-

Syntax

```
date - days
date1 - date2
timestamp - days
timestamp1 - timestamp2
```

```
time - minutes
time1 - time2
```

Description

Subtracts a number of days from a date or a timestamp. Subtracts a number of minutes from a time value. Calculates the number of days between two dates or timestamps. Calculates the number of minutes between two time values.

Examples

```
CURRENT_DATE - 1 --Yesterday
CURRENT_TIMESTAMP - 1 --24 hours ago
CURRENT_DATE - DATE'1/1/2006' --Number of days since the beginning of
2006
CURRENT_TIME - 60 --One hour ago
CURRENT_TIME - TIME'12:00 pm' --Number of minutes since noon (may be
negative)
```

CURRENT_DATE

Syntax

```
CURRENT_DATE
```

Description

Returns the date of the current day according to your system (local time).

CURRENT_TIME

Syntax

```
CURRENT_TIME
```

Description

Returns the time of the current millisecond according to your system (local time).

CURRENT_TIMESTAMP

Syntax

```
CURRENT_TIMESTAMP
```

Description

Returns the timestamp of the current millisecond (i.e. *CURRENT_DATE* and *CURRENT_TIME* together) according to your system (local time).

DATETIMESTR

Syntax

```
DATETIMESTR(TimeStamp, Precision)
```

Description

Calculates a string representation of the time stamp in the current locale. Precision is 2 for minutes, 3 for seconds and 4 for milliseconds.

EXTRACT

Syntax

```
EXTRACT(kind FROM date)
```

Description

Calculates a value from date. *Kind* is one of these:

YEAR	Returns the year.
MONTH	Returns the month.
DAY	Returns the day.

WEEKDAY	Returns the day of the week. 1 for Monday, 2 for Tuesday etc.
WEEKDAYNAME	Returns the name of the day of the week in the current locale.
WEEK	Returns the number of the week in the year according the ISO standard.
HOUR	Returns the hour.
MINUTE	Returns the minute.
SECOND	Returns the second.
MILLISECOND	Returns the millisecond.

Examples

```
EXTRACT(DAY FROM CURRENT_DATE)
EXTRACT(HOUR FROM CURRENT_TIME)
EXTRACT(SECOND FROM CURRENT_TIMESTAMP)
EXTRACT(WEEKDAYNAME FROM CURRENT_DATE)
EXTRACT(MILLISECOND FROM CURRENT_TIME)
EXTRACT(WEEK FROM CURRENT_TIMESTAMP)
```

MAKEDATE

Syntax

```
MAKEDATE(year, month, day)
```

Description

Returns the date value for the given date.

Example

```
SELECT * FROM MyTable WHERE Abs(Today - MakeDate(EXTRACT(YEAR FROM
CURRENT_DATE), EXTRACT(MONTH FROM Birthday), EXTRACT(DAY FROM
Birthday))) < 7
```

MAKETIMESTAMP

Syntax

```
MAKETIMESTAMP(year, month, day, hour, minute, second, millisecond)
```

Description

Returns the time stamp value for the given datetime.

MAKETIME

Syntax

```
MAKETIME(hour, minute, second, millisecond)
```

Description

Returns the time value for the given time.

TIMESTR

Syntax

```
TIMESTR(time, precision)
```

Description

Calculates a string representation of the time value in the current locale. Precision is 2 for minutes, 3 for seconds and 4 for milliseconds.

See also

General Functions and Operators

1.3.4.4.14 Aggregation Functions

This is a list of aggregation functions that can be used in TurboSQL.

AVG

Syntax

```
AVG(column_reference)
```

Description

Calculates the average of the values in the column. The argument must be a numeric type. The result is always a a FLOAT.

COUNT

Syntax

```
COUNT(*|column_reference)
```

Description

Calculates the number of rows in the column. The argument can be of any type. The result is always a BIGINT.

Examples

```
COUNT(*)
COUNT(NAME)
```

MAX

Syntax

```
MAX(column_reference)
```

Description

Calculates the maximum of the values in the column. The argument must be a numeric type or a date/time type. The result is a super-type of the argument type.

MIN

Syntax

```
MIN(column_reference)
```

Description

Calculates the minimum of the values in the column. The argument must be a numeric type or a date/time type. The result is a super-type of the argument type.

STDDEV

Syntax

```
STDDEV(column_reference)
```

Description

Calculates the standard deviation of the values in the columns. The argument must be numeric type. The result is always a FLOAT.

Example

```
SELECT AVG(Value), STDDEV(Value) FROM Values
```

Compatibility Information

This function is only available in TurboDB Managed.

SUM

Syntax

```
SUM(column_reference)
```

Description

Calculates the sum of the values in the column. The argument must be a numeric type. The result is a super-type of the argument type.

See also

General Functions and Operators
Arithmetic Functions and Operators
String Functions and Operators
Date and Time Functions and Operators
Aggregation Functions
Miscellaneous Functions and Operators
User-defined Aggregates

1.3.4.4.15 Miscellaneous Functions and Operators

This is a list of miscellaneous functions and operators that can be used in TurboSQL.

CONTAINS

Syntax

```
CONTAINS(full-text-search-expression IN table-name.*)
CONTAINS(full-text-search-expression IN column-name1, column-name2,
column-name3, ...)
```

Description

Evaluates to true, if the row satisfies the full-text search-expression. In the second variant, the full-text search-expression must be satisfied on the given column subset.

Example

If you are searching for a row where the word 'computer' is contained both in the column Category and in the column Name, you write:

```
SELECT * FROM Devices WHERE CONTAINS('computer' IN Category) AND
CONTAINS('computer' IN Name)
```

Compatibility Information

The second variant is only available in TurboDB Win.

NEWGUID

Syntax

```
NEWGUID
```

Description

Creates a new Globally Unique Identifier, like for example {2A189230-2041-44A6-87B6-0AFEE240F09E}.

Example

```
INSERT INTO TABLEA ("Guid") VALUES(NEWGUID)
```

CURRENTRECORDID

Syntax

```
CURRENTRECORDID(table_name)
```

Description

Returns the last used record id of the given table. Using this function, it is possible to enter linked records in multiple tables within one compound statement.

NULLIF

Syntax

```
NULLIF(value1, value2)
```

Description

Returns NULL if *value1* and *value2* are equal and *value1* if they are not.

Compatibility Information

This function is only supported in TurboDB Managed.

See also

General Functions and Operators
Arithmetic Functions and Operators
String Functions and Operators
Date and Time Functions and Operators
Aggregation Functions
Miscellaneous Functions and Operators

1.3.4.4.16 Table Operators

TurboSQL supports the following operators to combine table rows. They all follow the standard SQL specification:

JOIN

Syntax

```
table_reference [INNER | LEFT OUTER | RIGHT OUTER | OUTER] JOIN
table_reference
```

Samples

```
SELECT * FROM A JOIN B ON A.a = B.a
```

```
SELECT * FROM A LEFT OUTER JOIN B ON A.a = B.a
```

Description

Returns all row pairs of the two table references, for which the join condition holds.

UNION

Syntax

```
table_term UNION [ALL] table_term [CORRESPONDING BY column_list]
```

Samples

```
SELECT * FROM TABLE A UNION SELECT * FROM TABLE B
```

Description

Returns all rows from the two table terms. The result set is unique if all is not specified. The two table terms must have compatible columns.

EXCEPT

Syntax

```
table_term EXCEPT [ALL] table_term CORRESPPONDING [BY column_list]
```

Samples

```
SELECT * FROM TABLE A EXCEPT SELECT * FROM TABLE B
```

Description

Returns all rows from the first table term that do not exist in the second one. The result set is unique if all is not specified. The two table terms must have compatible columns.

INTERSECT

Syntax

```
table_primitive INTERSECT table_primitive CORRESPONDING [BY column_list]
```

Samples

```
SELECT * FROM TABLE A INTERSECT [ALL] SELECT * FROM TABLE B
```

Description

Returns all rows that exist in both the first and the second table term. The result set is unique if all is not specified. The two table terms must have compatible columns.

1.3.4.4.17 Sub-Queries

Search-conditions within SELECT, INSERT and UPDATE statements may contain embedded queries, which can be compared to the main query via one of the following operators. Furthermore, a select expression in parenthesis can be used everywhere an expression is expected. TurboSQL allows for uncorrelated sub-selects as well as for correlated ones.

IN

Checks whether the value of an expression can be found in the result set of the sub-query.

Example

```
select * from SALESINFO
where customerName in (
  select name from CUSTOMER where state = 'CA'
)
```

Selects all sales to customers from California and is basically the same as

```
select * from SALESINFO join CUSTOMER on customerName = name
where state = 'CA'
```

EXISTS

Checks whether the sub-query contains at least one row.

Example

```
select * from SALESINFO
where exists (
  select * from CUSTOMER
  where name = SALESINFO.customerName and state = 'CA'
)
```

Retrieves the same result as the first example. Note however that this time the sub-query contains a column reference to the outer query. This is called correlated sub-query.

ANY/SOME

Checks whether there is at least one row in the result of the sub-query, which satisfies the search-condition.

Example

```
select * from SALESINFO
where amount > any (
  select averageAmount from CUSTOMER
  where name = SALESINFO.customerName
)
```

Retrieves all sales bigger than the average for the respective customer.

ALL

Checks whether the search-condition is satisfied for all rows in the result of the sub-query.

Example

```
select * from SALESINFO
where amount > all (
  select averageAmount from CUSTOMER
  where state = 'CA'
)
```

Retrieves the sales bigger than the average volume for each single customer in California.

Sub-Query as Expression

A select expression in parenthesis can be used as a scalar expression. The type of the scalar expression is the type oft the first column in the result set. The value of the scalar expression is the value of the first column in the first row. If the result set has no column, the expression is invalid. If it has no rows, the result value is NULL.

Examples

```
select * from [TableB] where C1 like (select C2 from TableB) || '%'
```

```
set A = (select Count(*) from TableA)
```

Compatibility Information

The use of a sub-select as an expression is only available in TurboDB Managed.

See also

WHERE

1.3.4.4.18 Full-Text Search

Full-text search is the search for an arbitrary word in a table row. This kind of search is especially useful for memo and wide memo fields, where searching with conventional operators and functions does not deliver the expected result or takes too long.

Full-text search in TurboDB has two restrictions:

- There must be a full-text index on the table to use full-text searching capabilities.
- One full-text search-condition always refers to a single table. (There can be multiple full-text search-conditions in the same where clause however.)

The basis of a full-text index is the dictionary, which is a normal database table with a certain schema. It holds the information on indexed words, excluded words, word relevance etc. Once the dictionary exists, it can be used for any number of full-text indexes on one or on multiple tables.

As of TurboDB 5, full-text search-conditions are embedded in the WHERE clause of the statement:

```
select * from SOFTWARE join VENDOR on SOFTWARE.VendorId = VENDOR.Id
where VENDOR.Country = 'USA' and (contains('office' in SOFTWARE.*) or
contains('Minneapolis' in VENDOR.*))
```

A simple full-text search condition looks like this:

```
contains('office -microsoft' in SOFTWARE.*)
```

which is true, if any of the fields of the default full-text index of table SOFTWARE contains the word *office* but not the word *microsoft*. If the query refers to only one table, this can also be written as

```
contains('office -microsoft' in *)
```

If the full-text search-expression contains more than one word without the hyphen, TurboDB searches for rows that contain all the given words. Therefore

```
contains('office microsoft' in SOFTWARE.*)
```

will find rows, that contain both the word *office* and *microsoft* in any of the fields of the default

full-text index of the table.

Words separated by a plus sign are searched for alternatively. The predicate

```
contains('office star + open' in SOFTWARE.*)
```

finds rows containing the word *office* plus either the word *star* or the word *open* (or both).

A full-text search-condition can be evaluated also on just a sub-set of the indexed columns:

```
contains('office' in SOFTWARE.Name, SOFTWARE.Category)
```

Will find all rows where the word *office* occurs either in column *Name* or column *Category*. If an additional column *Description* is also indexed, a row will not be returned if it only occurs in that column but not in *Name* or *Category*. In general, the full-text search-condition is evaluated on the union of indicated columns. So if there is a excluding term in it, like in

```
contains('office -microsoft' in Name, Category)
```

the word microsoft must appear neither in *Name* nor in *Category*, if it appears in *Description* the row can nevertheless be part of the result set.

Notes

Full-text indexes can be created with the CREATE FULLTEXTINDEX TurboSQL statement, with one of the database management tools (like TurboDB Viewer) or with the appropriate functions of the respective TurboDB access components.

The full-text searching technology has changed between TurboDB level 3 and level 4 tables. The new implementation is much faster and allows for maintained full-text indexes as well as for row relevance. It is strongly recommended to use level 4 tables, when working with full-text searching capabilities. The old full-text search will eventually be removed.

1.3.4.5 Data Definition Language

TurboSQL supports these statements and data types as part of the Data Definition Language:

Statements

CREATE TABLE

ALTER TABLE

CREATE INDEX

CREATE FULLTEXTINDEX

UPDATE INDEX/FULLTEXTINDEX

DROP

Data types

TurboSQL data types

1.3.4.5.1 CREATE TABLE Statement

Creates a new table within the current database.

Syntax

```
CREATE TABLE table_reference
[LEVEL level_number]
[ENCRYPTION encryption_algorithm [PASSWORD password]]
[COLLATE collation_name]
(column_definition | constraint_definition [, column_definition |
constraint_definition] ...)
```

where a *column_definition* looks like this:

```
column_reference data_type [NOT NULL] [DEFAULT expression | SET
expression]
```

(see column data types for detailed information)

and a *constraint_definition* like this:

```
PRIMARY KEY (column_reference [, column_reference]...) |
UNIQUE (column_reference [, column_reference]...) |
[CONSTRAINT constraint_name] CHECK (search_condition) |
FOREIGN KEY (column_reference [, column_reference]...)
REFERENCES table_reference (column_reference [, column_reference]...)
[ON UPDATE NO ACTION | CASCADE]
[ON DELETE NO ACTION | CASCADE]
```

Description

Use the *CREATE TABLE* command when you want to add a new table to the database.

```
CREATE TABLE MyTable (
  OrderNo AUTOINC,
  OrderId CHAR(20) NOT NULL,
  Customer LINK(Customer) NOT NULL,
  OrderDate DATE NOT NULL DEFAULT Today,
  Destination ENUM(Home, Office, PostBox),
  PRIMARY KEY(OrderNo),
  CHECK(LEN(OrderId) > 3),
  FOREIGN KEY(Customer) REFERENCES Customer(CustNo) ON DELETE CASCADE ON
UPDATE NO ACTION
)
```

To define a password, a key and a language, add the appropriate keywords:

```
CREATE TABLE MyTable
  ENCRYPTION 'Blowfish' PASSWORD 'u(i,iUklah'
  LANGUAGE 'ENU'
  (Name CHAR(20))
```

The encryption algorithms currently supported are described in "Data Security". If an encryption algorithm is given, the password must be indicated as well.
The level number is used for backward compatibility. If it is omitted the most current table format will be created.

The other clauses have their standard SQL meaning. Note that TurboSQL does not yet support the *set null* and the *set default* actions for the foreign key present in the SQL standard.

DEFAULT defines the default value for the column, which is assigned when a new row is created. Available on table level 4 or higher.

SET defines an expression used to calculate the column value each time the row is updated. *SET* columns are read-only.

See also

Column Data Types

1.3.4.5.2 ALTER TABLE Statement

Modifies columns and column types of an existing table.

Syntax

```
ALTER TABLE table_reference
[LEVEL level_number]
[ENCRYPTION encryption_algorithm]
[PASSWORD password]
[COLLATE collation_name]
DROP column_reference |
DROP CONSTRAINT constraint_name |
ADD column_definition |
ADD CONSTRAINT constraint_definition |
RENAME column_reference TO column_reference |
```

```
MODIFY column_definition
```

Description

The ALTER TABLE command enables you to modify the structure of an existing table. Please find the description for *column_definition* and *constraint_definition* in the topic about the *CREATE TABLE statement*. There are six different options:

Delete an existing column with *DROP*:

```
ALTER TABLE Orders DROP Destination
```

The *column_reference* must refer to an existing column.

Delete an existing constraint with *DROP*:

```
ALTER TABLE Orders DROP CONSTRAINT sys_Primary
```

Add a new column with *ADD*:

```
ALTER TABLE Orders ADD Date_of_delivery DATE
```

The name of the new column must not exist before.

Add a new constraint with *ADD*, which can be either a named constraint or an unnamed constraint:

```
ALTER TABLE Orders ADD CONSTRAINT RecentDateConstraint CHECK
(Date_of_delivery > 1.1.2000)
```

```
ALTER TABLE Orders ADD FOREIGN KEY (Customer) REFERENCES Customer
(CustNo)
```

Modify the name of an existing column with *RENAME*:

```
ALTER TABLE Orders RENAME Date_of_delivery TO DateOfDelivery
```

The first *column_reference* is the name of an existing column, the second is the new name of this column. Renaming a column keeps the data within the column intact.

Modify the column type of an existing column with *MODIFY*:

```
ALTER TABLE Orders MODIFY DateOfDelivery TIMESTAMP
```

The *column_reference* must refer to an existing column. You may change the column type to any one of the available column types. The column data is kept as far as possible.

The parameters *level_number*, password, key and language have the same meaning as in the *CREATE TABLE* statement. If password and key are omitted, the current settings are kept. To remove the encryption, set it to *NONE*.

This statement removes encryption from a table:

```
ALTER TABLE Orders ENCRYPTION None
```

Note: If the password or the encryption mode are to be changed, both the password and the encryption must be indicated for security reasons.

It is possible to combine multiple changes in any order within one single command:

```
ALTER TABLE Orders
  ADD Date_of_delivery DATE,
  DROP Destination,
  ADD DeliveryAddress CHAR(200),
  RENAME Customer TO CustomerRef
```

Note: *RENAME* and *MODIFY* are proprietary extensions to SQL-92.

Compatibility Information

The *COLLATE* clause is supported from table level 6 on.

The *SET* clause in the column definition is only supported for table levels greater than 3.

See also

Column Data Types

1.3.4.5.3 CREATE INDEX Statement

Creates a new index for an existing table.

Syntax

```
CREATE [UNIQUE] INDEX index_reference ON table_reference
(column_reference [ASC|DESC] [,column_reference [ASC|DESC] ...] )
```

Description

Indexes are used to speed up query execution. Use the *CREATE INDEX* command to add a new index to an existing table:

```
CREATE INDEX OrderIdIdx ON Orders (OrderId ASC)
```

You may create multi-level hierarchical indexes as well:

```
CREATE INDEX TargetDateIdx ON Orders (DeliveryDate DESC, OrderId)
```

Use `UNIQUE` to create an index that raises an error if rows with duplicate column values are inserted. By default, indexes are not unique.

Note

The ASC and DESC attribute at column level is a proprietary extension to SQL-92.

1.3.4.5.4 CREATE FULLTEXTINDEX Statement

Creates a new full-text index for an existing table.

Syntax

```
CREATE FULLTEXTINDEX index_reference ON table_reference
(column_reference [, column_reference ...])
DICTIONARY table_reference [CREATE] [UPDATE]
[SEPARATORS '<separating chars>']
```

Description

A full-text index enables searching with full-text search-conditions like the *CONTAINS* predicate. The column references are a list of table columns of all types with the exception of blob columns including memos and wide memos. Full-text indexes need an additional database table, which contains the list of indexed words, the dictionary.

The dictionary table can be created by this statement or explicitly. If *CREATE* is not indicated, the statement expects an existing dictionary table with the following characteristics:

- First column is a *VARCHAR* or *VARWCHAR* of arbitrary length. This column determines the possible search-words. If words are longer than this column allows, they are cut.
- Second column is of type *BYTE*. It contains the global relevance of this word.
- Other columns may or may not follow according to the needs of the application.
- There must be a column of type *AUTOINC* to identify the words. The indication of this *AUTOINC* column must be the first column of the table.

If the CREATE clause is included, the statement creates a new dictionary table with a first column as *VARCHAR(20)*.

If UPDATE is included, words that are not found in the dictionary table are added to it. If *UPDATE* is not included, only words from the dictionary table are indexed and can be found in searches.

If the *SEPARATORS* clause is included, you can define how the index splits text in words. For example, sometimes hyphens should be regarded as part of the word and sometimes as separators. The characters defined in *<separating chars>* replace the default separating chars, so you must indicate all desired separators including the obvious ones like space, comma, dot etc.

Note

Full-text search technology for tables up to level three is different from the one for tables starting from level four. Only full-text indexes for tables of level four and above support automatic

maintenance and relevance calculation.

1.3.4.5.5 DROP Statement

Delete a table or an index from the database.

Syntax

```
DROP TABLE table_reference
```

```
DROP INDEX table_reference.index_name
```

```
DROP FULLTEXTINDEX table_reference.index_name
```

Description

Use *DROP* to completely remove the database object and all appropriate files from the database.

Examples

```
DROP INDEX Orders.OrderIdIdx
DROP TABLE Orders
```

Warning

While an index deleted by error can be re-created easily, the data in a dropped table is gone and cannot be restored.

1.3.4.5.6 UPDATE INDEX/FULLTEXTINDEX Statement

Repairs an index or full-text index.

Syntax

```
UPDATE INDEX table_reference.index_name
```

```
UPDATE FULLTEXTINDEX table_reference.index_name
```

Description

If an index seems out-of-date (this can happen in the embedded version of TurboDB, if the client application crashes), the index can be re-built using this command. Use the asterisk * for the *index_reference* to update all indexes of a table.

Note

The UPDATE FULLTEXTINDEX statement is available only for the new maintained full-text indexes for level 4 tables.

Example

```
UPDATE INDEX MyTable.*; UPDATE FULLTEXTINDEX MyTable.*
```

1.3.4.5.7 TurboSQL Column Types

These are the column types supported by *TurboSQL*.

AUTOINC

Syntax

```
AUTOINC(indication) [UNIQUE]
```

TurboDB Column Type

AutoInc

Description

Integer field which receives a unique number from the database engine. Field values of link columns in dependent tables are displayed according to the indication, which is a string containing an index definition. (See "Automatic Linking".)

If UNIQUE is indicated, the indication is forced to be unique.

Example

```
AUTOINC('LastName, FirstName')
```

BIGINT

Syntax

```
BIGINT [NOT NULL]
```

TurboDB Column Type

BigInt

Description

An integral number between –2^63 and +2^63–1

Example

```
BIGINT DEFAULT 4000000000
```

BIT

Not supported in TurboSQL.

BOOLEAN

Syntax

```
BOOLEAN [NOT NULL]
```

TurboDB Column Type

Boolean

Description

Possible values are TRUE and FALSE

Example

```
BOOLEAN DEFAULT TRUE
```

BYTE

Syntax

```
BYTE [NOT NULL]
```

TurboDB Column Type

Byte

Description

An integral number between 0 and 255

Example

```
BYTE NOT NULL DEFAULT 18
```

CHAR

Syntax

CHAR(*n*) [NOT NULL] [COLLATE *collation-name*]

TurboDB Column Type

String

Description

Ansi string up to N characters long. 1 <= n <= 32767

Example

```
CHAR(40)
```

Compatibility Information

The COLLATE clause is supported from table level 6 on.

CURRENCY

Not supported in TurboSQL, use DOUBLE PRECISION or BIGINT.

DATE

Syntax

```
DATE [NOT NULL]
```

TurboDB Column Type

Date

Description

Date value between 1/1/1 and 12/31/9999

Example

```
DATE
```

DECIMAL

Not supported in TurboSQL, use DOUBLE PRECISION.

DOUBLE PRECISION

Syntax

```
DOUBLE [PRECISION][(p)] [NOT NULL]
```

TurboDB Column Type

Float

Description

Floating point number from 5.0e-324 to 1.7 x 10e308 with a precision of 12 signification digits. p is the number of displayed digits after the decimal point. 0 <= p <= 12.

Example

```
DOUBLE(4) NOT NULL
```

ENUM

Syntax

```
ENUM(value1, value2, value3, ...) [NOT NULL]
```

TurboDB Column Type

Enum

Description

Column that holds one of the enumeration values given stored as a number internally. The values must be valid identifiers up to 40 characters in length. There can be up to 15 values.

Example

```
ENUM(Red, Blue, Green, Yellow)
```

FLOAT

Not supported in TurboSQL, use DOUBLE PRECISION.

GUID

Syntax

```
GUID [NOT NULL]
```

TurboDB Column Type

Guid

Description

A universally unique identifier 16 bytes in size.

Example

```
GUID DEFAULT '12345678-abcd-abcd-efef-010101010101'
```

INTEGER

Syntax

```
INTEGER [NOT NULL]
```

TurboDB Column Type

Integer

Description

An integral number between –2.147.483.648 and +2.147.483.647

Example

```
INTEGER NOT NULL
```

LINK

Syntax

```
LINK(table_reference) [NOT NULL]
```

TurboDB Column Type

Link

Description

Holds value of *AutoInc* column of another table and such builds a one-to-many relationship. *Table_reference* is the name of referenced table. (See "Automatic Linking".)

Example

```
LINK(PARENTTABLE)
```

LONGVARBINARY

Syntax

```
LONGVARBINARY [NOT NULL]
```

TurboDB Column Type

Blob

Description

Long bit-stream containing arbitrary data up 2 GB

Example

```
LONGVARBINARY
```

LONGVARCHAR

Syntax

```
LONGVARCHAR [NOT NULL] [COLLATE collation-name]
```

TurboDB Column Type

Memo

Description

Long Ansi string of variable length up to 2 G characters

Example

```
LONGVARCHAR
```

Compatibility Information

The COLLATE clause is supported from table level 6 on.

LONGVARWCHAR

Syntax

```
LONGVARWCHAR [NOT NULL] [COLLATE collation-name]
```

TurboDB Column Type

WideMemo

Description

Long Unicode string of variable length up to 2 G characters

Example

```
LONGVARWCHAR
```

Compatibility Information

The COLLATE clause is supported from table level 6 on.

MONEY

Not supported in TurboSQL, use DOUBLE PRECISION or BIGINT.

NUMERIC

Not supported in TurboSQL, use DOUBLE PRECISION.

RELATION

Syntax

```
RELATION(table_reference)
```

TurboDB Column Type

Relation

Description

Holds any number of AutoInc values of another table. Used to create a many-to-many relationship. (See "Automatic Linking".)

Compatibility Information

Feature is not supported in TurboDB Managed 1.x.

Example

```
RELATION(PARENTTABLE)
```

TIME

Syntax

```
TIME[(p)] [NOT NULL]
```

TurboDB Column Type

Time

Description

Time of day with a precision of p, where p = 2 means minutes, p = 3 means seconds and p = 4 means milliseconds, the default being 3. The precision is available only in table level 4 and above; it is always set to 2 in tables up to level 3.

Example

```
TIME(4) DEFAULT 8:32:12.002
```

TIMESTAMP

Syntax

```
TIMESTAMP [NOT NULL]
```

TurboDB Column Type

DateTime

Description

Combined date and time with a precision of milliseconds between 1/1/1 12:00:00.000 am and 12/31/9999 11:59:59.999 pm

Examples

```
TIMESTAMP DEFAULT 23.12.1899_15:00:00
TIMESTAMP DEFAULT '5/15/2006 7:00:00'
```

VARCHAR

Syntax

```
VARCHAR(n) [NOT NULL] [COLLATE collation-name]
```

TurboDB Column Type

String

Description

Same as CHAR

Example

```
VARCHAR(40)
```

Compatibility Information

The COLLATE clause is supported from table level 6 on.

VARWCHAR

Syntax

```
VARWCHAR(n) [NOT NULL] [COLLATE collation-name]
```

TurboDB Column Type

WideString

Description

Same as WCHAR

Example

```
VARWCHAR(20) NOT NULL
```

Compatibility Information

The COLLATE clause is supported from table level 6 on.

SMALLINT

Syntax

```
SMALLINT [NOT NULL]
```

TurboDB Column Type

SmallInt

Description

An integral number between -32.768 and +32.767

Example

```
SMALLINT
```

WCHAR

Syntax

```
WCHAR(n) [NOT NULL] [COLLATE collation-name]
```

TurboDB Column Type

WideString

Description

Unicode string up N characters long. The actual field size in bytes is twice the number of characters. 1 <= n <= 32767

Example

```
WCHAR(1000) DEFAULT '-'
```

Compatibility Information

The COLLATE clause is supported from table level 6 on.

1.3.4.6 Programming Language

This feature is only available in TurboDB Managed.

TurboSQL provides language elements to create routines that can be called from SQL commands.

User-defined functions

Functions are used to simplify SQL commands and to provide additional functionality. They can be coded either in TurboSQL or be imported from a .NET assembly. Functions can only have input parameters and always calculate a return type. They must not have side-effects. Functions are managed using the CREATE FUNCTION and DROP FUNCTION statements. Functions can be used for computed indexes, computed columns and checks.

User-defined procedures

Procedures are used to call complex sequences of SQL statements with a single statement. For example, using a procedure, multiple rows can be updated in different tables in one step. Procedures can be implemented either in TurboSQL or be imported from a .NET assembly. Procedures are managed using the CREATE PROCEDURE and DROP PROCEDURE statements.

User-defined aggregates

Aggregates compute accumulated values in result set groups. They can be used for example to compute the 2nd order maximum, the standard deviation or other accumulated values from a grouped result set. Aggregates are implemented in a .NET assembly and managed with the CREATE AGGREGATE and DROP AGGREGATE statements.

Statements

CREATE FUNCTION Statement

CREATE PROCEDURE Statement

CREATE AGGREGATE Statement

DROP FUNCTION/PROCEDURE/AGGREGATE Statement

DECLARE Statement

SET Statement

WHILE Statement

IF Statement

CALL Statement

Other topics

Exchanging parameters with .NET Assemblies

1.3.4.6.1 CALL Statement

Syntax

```
CALL procedure_name([argument, ...])
```

Description

Executes a stored procedure with the given arguments.

Sample

```
CALL LogLine('This is a test statement')
```

Compatibility Information

This statement is only available in TurboDB Managed.

1.3.4.6.2 CREATE FUNCTION Statement

Syntax

```
CREATE FUNCTION function_name([:parameter_name data_type]...) RETURNS
data_type AS RETURN expression
```

```
CREATE FUNCTION function_name([:parameter_name data_type]...) RETURNS
data_type AS BEGIN statement [statement]... END
```

```
CREATE FUNCTION function_name([:parameter_name data_type]...) RETURNS
data_type AS EXTERNAL NAME [namespace_name.class_name].method_name,
assembly_name
```

Description

A function can be called wherever a scalar value is expected, in select elements, in search conditions and all other kinds of expressions.

namespace_name.class_name refers to a public class in the assembly.

method_name refers to the name of a static public function in the assembly. The function must not be overloaded and the parameters must fit the parameters of the TurboSQL function (see "Exchanging Parameters with .NET Assemblies").

Samples

```
CREATE FUNCTION LastChar(:S WCHAR(1024)) RETURNS WCHAR(1) AS
RETURN SUBSTRING(:S FROM CHAR_LENGTH(:s) FOR 1)
```

```
CREATE FUNCTION Replicate(:S WCHAR(1024), :C INTEGER)RETURNS WCHAR(1024)
AS BEGIN
     DECLARE :I INTEGER
     DECLARE :R WCHAR(1024)
     SET :I = 0
     SET :R = ''
     WHILE :I < :C BEGIN
          SET :R = :R + :S
          SET :I = :I + 1
     END
     RETURN :R
END
```

```
CREATE FUNCTION CubicRoot(:X FLOAT) RETURNS FLOAT AS
EXTERNAL NAME [MathRoutines.TurboMath].CubicRoot,MathRoutines
```

Compatibility Information

This statement is only available in TurboDB Managed.

1.3.4.6.3 CREATE PROCEDURE Statement

Syntax

```
CREATE PROCEDURE function_name([:parameter_name data_type]...) AS
statement
```

```
CREATE PROCEDURE function_name([:parameter_name data_type]...) AS BEGIN
statement [statement]... END
```

```
CREATE PROCEDURE function_name([:parameter_name data_type]...) AS
EXTERNAL NAME [namespace_name.class_name].method_name,assembly_name
```

Description

namespace_name.class_name refers to a public class in the assembly.

method_name refers to the name of a static public function in the assembly. The function must not be overloaded and the parameters must fit the parameters of the TurboSQL function (see "Exchanging Parameters with .NET Assemblies").

Samples

```
CREATE PROCEDURE Insert(:LastName WCHAR(40), :Salary INTEGER, :
Department WCHAR(40) AS BEGIN
     INSERT INTO Departments ([Name]) VALUES(:Department)
     INSERT INTO Employees (LastName, Salary, Department) VALUES(:
LastName, :Salary, :Department)
END
```

The following example requires a static public method *Send* of a public class *SmtpClient* in the namespace *Email* in an assembly called *Email.dll*. The assembly must be located in the same directory as is the database (tdbd) file.

```
CREATE PROCEDURE SendEmail(:Address WCHAR(100), :Subject WCHAR(100), :
Content WCHAR(100)) AS
EXTERNAL NAME [Email.SmtpClient].Send,Email
```

Compatibility Information

This statement is only available in TurboDB Managed.

1.3.4.6.4 CREATE AGGREGATE Statement

Syntax

```
CREATE AGGREAGATE aggregate_name([:parameter_name data_type]...) RETURNS
data_type AS EXTERNAL NAME [namespace_name.class_name],assembly_name
```

Description

namespace_name.class_name refers to a public class in the assembly. This class must have a default constructor and implement three methods:

```
[C#]
public <class_name>()
public void Init()
public void Accumulate(<data_type>)
public <data_type> Terminate()
```

The *data_type* must fit the parameters of the TurboSQL function (see "Exchanging Parameters with .NET Assemblies").

When a user-defined aggregate is used in a SQL statement, e.g. *SELECT MAX2(Salary) FROM Employees*, the execution engine creates an instance of the CLR aggregation class. At the beginning of each group it calls the *Init* function. After this, it calls the *Accumulate* function for each row in the group in the order defined by the GROUP BY clause. After all rows of the group have been accumulated, the execution engine calls the *Terminate* function to retrieve the result. Any resources allocated can be disposed of in the *Terminate* function.

Samples

The C# code for an aggregate that computes the second order maximum looks like this.

```
namespace MathRoutines {

     public class SecondOrderMax {

          public SecondOrderMax() { }

          public void Init() {
               max = null;
               max2 = null;
          }

          public void Accumulate(double? x) {
               if (max == null || x > max) {
                    if (max != null)
                         max2 = max;
                    max = x;
               } else if (max2 == null || x > max2)
                    max2 = x;
          }

          public double? Terminate() {
               return max2;
          }

          private double? max;
          private double? max2;
     }
}
```

And this is how you import the aggregate into TurboSQL.

```
CREATE AGGREGATE Max2(:x DOUBLE) RETURNS DOUBLE AS
EXTERNAL NAME [MathRoutines.SecondOrderMax],MathRoutines
```

Compatibility Information

This statement is only available in TurboDB Managed.

1.3.4.6.5 DROP FUNCTION/PROCEDURE/AGGREGATE Statement

Syntax

```
DROP FUNCTION | PROCEDURE | AGGREGATE
```

Description

Use DROP to remove the function, procedure or aggregate from the database.

Compatibility Information

This statement is only available in TurboDB Managed.

1.3.4.6.6 DECLARE Statement

Syntax

```
DECLARE :variable_name data_type
```

Samples

```
DECLARE :i INTEGER
DECLARE :s CHAR(300)
```

Compatibility Information

This statement is only available in TurboDB Managed.

1.3.4.6.7 IF Statement

Syntax

```
IF search_condition THEN if_statement [ELSE else_statement]
```

Description

Executes the *if_statement* if and only if *search_condition* evaluates to True. If *search_condition* does not return True and *else_statement* is given, it is executed.

Sample

```
IF :s <> '' THEN SET :s = :s + ', '
```

```
IF :a > 0 THEN BEGIN
     SET :r = SQRT(:a)
END ELSE BEGIN
     SET :r = SQRT(-:a)
END
```

Compatibility Information

This statement is only available in TurboDB Managed.

1.3.4.6.8 SET Statement

Syntax

```
SET :variable_name = expression
```

Description

Assigns a new value to the variable.

Example

```
SET :LastName = 'Miller'
```

Compatibility Information

The SET statement is only available in TurboDB Managed.

1.3.4.6.9 WHILE Statement

Syntax

```
WHILE search_condition statement
```

Description

Executes statement as long as *search_condition* evaluates to True.

Sample

```
DECLARE :I INTEGER
WHILE :I < 100 SET :I = :I + 1
```

```
DECLARE :I INTEGER
WHILE :I < 100 BEGIN
     SET :I = :I + 1
END
```

Compatibility Information

This statement is only available in TurboDB Managed.

1.3.4.6.10 Exchanging Parameters with .NET Assemblies

Assemblies that are used for external functions, procedures or aggregates must be located in the same directory as the database file.

When calling methods from .NET assemblies, parameters are mapped from TurboSQL to CLR according to the table below. Regarding this table, there is a difference between functions on one hand and procedures and aggregates on the other hand.

Functions have only input parameters and are evaluated to NULL, if one of its arguments is NULL. In this case, the CLR method is not executed at all. Therefore, NULL will never be passed to user-defined CLR functions and the CLR method must be declared with the non-nullable CLR types only. For example,

```
CREATE FUNCTION Log2(:x DOUBLE NOT NULL) RETURNS DOUBLE NOT NULL AS
EXTERNAL NAME [MyNamespace.MyClass].MyMethod,MyAssembly
```

corresponds to this definition in C#:

```
public class MyClass {
     static public double MyMethod(double x) {...}
}
```

Because the argument is not nullable, *Log2* can never be used on nullable arguments. However, if declared like this

```
CREATE FUNCTION Log2(:x DOUBLE) RETURNS DOUBLE AS EXTERNAL NAME
[MyNamespace.MyClass].MyMethod,MyAssembly
```

the same definition as above is valid in C#. When *Log2* is called with a NULL argument, the execution engine will return NULL without calling the CLR method.

This rule only holds for functions; CLR procedures and CLR aggregates are always called, even if one of the arguments is NULL. Therefore, the parameters and the return type of the CLR definition must be able to transport the NULL value. TurboSQL does this by passing a *null* value (*Nothing* in Visual Basic), which is obvious for CHAR types and array types like *LONGVARBINARY*. In order to be able to do this with value types like *Int64* or *DateTime*, the nullable wrapper must be used.

```
CREATE PROCEDURE TestProc(:x DOUBLE) AS EXTERNAL NAME [MyNamespace.
MyClass].MyMethod,MyAssembly
```

is therefore mapped by

```
public class MyClass {
     static public void MyMethod(Nullable<double> x) {...}
}
```

or

```
public class MyClass {
     static public void MyMethod(double? x) {...}
}
```

or in Visual Basic:

```
Public Class MyClass
     Public Shared Sub MyMethod(X As Nullable(Of Double))
          ...
     End Sub
End Class
```

For more information on nullable wrappers, search MSDN for the *Nullable* class.

TurboSQL type	Non-Nullable CLR type	Nullable CLR type
BYTE	*System.Int64*	*Nullable<System.Int64>*
SMALLINT	*System.Int64*	*Nullable<System.Int64>*
INTEGER	*System.Int64*	*Nullable<System.Int64>*

BIGINT	*System.Int64*	*Nullable<System.Int64>*
FLOAT	*System.Double*	*Nullable<System.Double>*
AUTOINC	*System.Int64*	*Nullable<System.Int64>*
BOOLEAN	*System.Int64 (0 corresponds to false, all other values to true)*	*Nullable<System.Int64>*
ENUM	*System.Int64*	*Nullable<System.Int64>*
GUID	*System.Guid*	*Nullable<System.Guid>*
LINK	*System.Int64*	*Nullable<System.Int64>*
RELATION	*System.Int64[]*	*System.Int64[]*
CHAR(X), VARCHAR (X)	*System.String*	*System.String*
WCHAR(X), VARWCHAR(X)	*System.String*	*System.String*
LONGVARCHAR	*System.String*	*System.String*
LONGVARWCHAR	*System.String*	*System.String*
LONGVARBINARY	*System.Byte[]*	*System.Byte[]*
TIME	*System.DateTime*	*Nullable<System.DateTime>*
DATE	*System.DateTime*	*Nullable<System.DateTime>*
TIMESTAMP	*System.DateTime*	*Nullable<System.DateTime>*

Compatibility Information

This statement is only available in TurboDB Managed.

1.3.5 TurboDB Products and Tools

For developers using TurboDB there is a set of additional tools:

TurboDB Viewer Visual tool for managing TurboDB database tables and indexes, editing database tables. Included in TurboDB packages.

TurboDB Pilot SQL console for managing and working with level 5 single-file databases.

dataweb Compound File Explorer: Visual tool for managing the storage objects within a dataweb compound file like e.g. a TurboDB single-file database.

TurboDB Workbench: Console application for creating, indexing, altering TurboDB tables. Included in TurboDB Components package.

TurboDB Data Exchange: Importing and exporting records to and from TurboDB tables.

TurboDB Server: Database server for TurboDB. Allows up to 100 concurrent sessions on the same database.

TurboDB Studio: Complete RAD tool for developing Windows applications with TurboDB Engine.

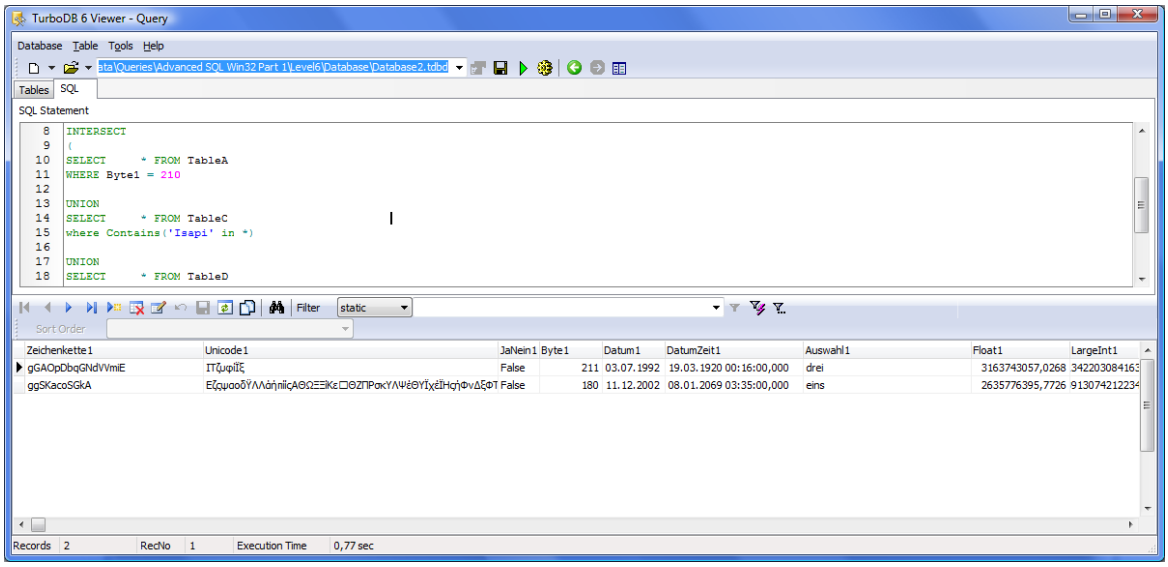
More information on TurboDB products and tools is to be found on the dataweb homepage www.dataweb.de.

1.3.5.1 TurboDB Viewer

TurboDB Viewer is a visual tool for managing tables and indexes and for editing tables. Here is what you can do using TurboDB Viewer:

- View and edit tables.
- Execute SQL queries.
- Create, alter and maintain database tables.
- View and define checks and foreign keys.
- Create and remove indexes.
- Create and remove full-text indexes.

TurboDB Viewer is included in all TurboDB Editions. Since it is itself written using the native version of TurboDB, it supports all table levels and features.

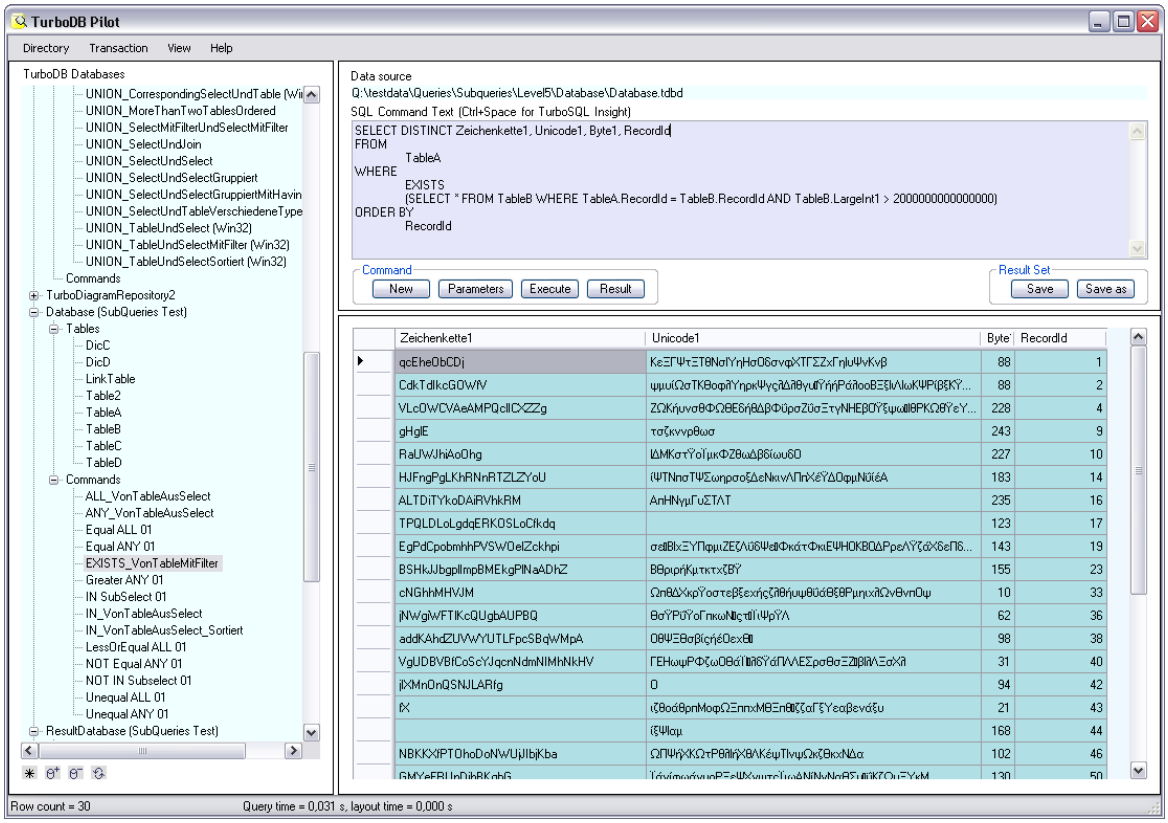


When using TurboDB Viewer to create databases for TurboDB Managed, you must however be careful to not use features, that are unsupported by TurboDB Managed. For TurboDB Managed you may want to use TurboDB Pilot, which is based on the managed database engine and is strictly SQL-based.

1.3.5.2 TurboDB Pilot

TurboDB Pilot is an SQL console for TurboDB level 5 databases written with the managed database engine. It is a complete tool for managing TurboDB level 5 databases.

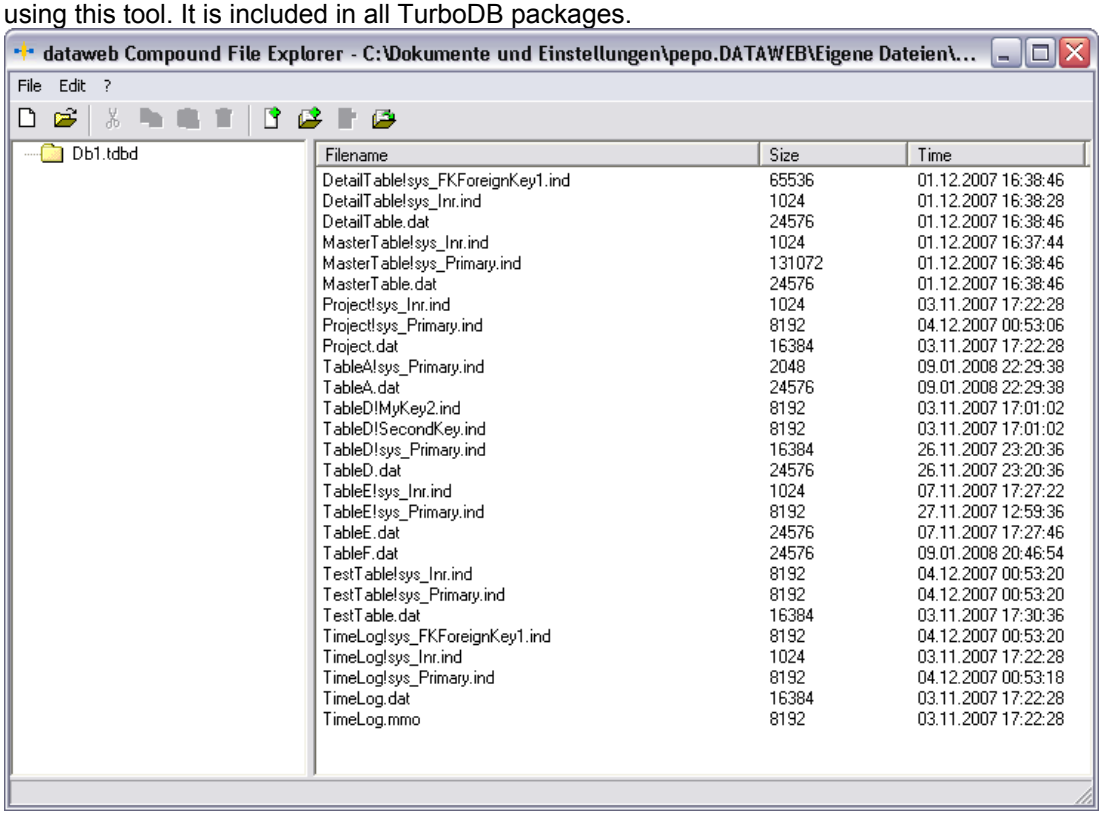
- Creating and altering tables
- Performing queries with parameters
- Updating databases
- Examining schema information



TurboDB Pilot stores a directory of all databases in your system and therefore makes it easy to retrieve them.

1.3.5.3 dataweb Compound File Explorer

This is the managing tool for dataweb's compound file technology. Since a TurboDB single-file database is a compound file, you can view and edit the storage objects of a single-file database using this tool. It is included in all TurboDB packages.



1.3.5.4 TurboDB Workbench

tdbwkb is a small text-based free-ware tool for managing TurboDB tables. It is available for Windows and Linux and can be downloaded at http://www.turbodb.de/. *tdbwkb* offers commands for

- Creating new tables
- Modifiying existing tables
- Show the table structure of existing tables
- Creating indexes for a table
- Deleting indexes for a table
- Repair a table and its indexes
- Deleting a table
- Switching between different databases

Running *tdbwkb* will show the copyright and the *tdbwkb* prompt where you can enter the different commands. Enter *help* to show a list of available commands.

Here is a sample *tdbwkb* session to illustrate the available features.

```
home/usr1>tdbwkb
dataweb Turbo Database Workbench Version 4.0.1 (TDB 6.1.6)
Copyright (c) 2002-2003 dataweb GmbH, Aicha, Germany
Homepage http://www.dataweb.de, Mail dataweb Team
```

```
Type 'help' to get a list of available commands.

tdbwkb> help
  Abbreviations are not allowed. The commands are:
  altertable    Modifies an existing table.
  bye           Ends the tdbwkb session.
  cd            Changes the current directory.
  debug         Toggles debug mode. (Debug mode prints log messages.)
  delindex      Deletes an index from a table.
  deltable      Deletes all files of a table.
  help          Prints this list of commands.
  newftindex    Create a new full text index for a table.
  newindex      Creates a new index for a table.
  newtable      Creates a new table.
  pwd           Prints current working directory.
  show          Shows a rough preview of the table.
  switchdb      Opens another database.
  rename        Renames a table.
  repair        Rebuilds a table and all its indexes.
  tableinfo     Shows the description of a table.
  Type help <cmd> to get more specific help for a command.
  Note: You may also use tdbwkb in batch mode by appending the command
directly
  to the call. Example:
  tdbwkb tableinfo mytable

tdbwkb> newtable animals
S40Name,A'Land,Water,Air'Area,PImage,MDescription,N'Name'RecordId
Creating table animals.dat with these columns:
  1 S40  Name
  2 A    Area, Values = Land,Water,Air
  3 P    Image
  4 M    Description
  5 N    RecordId
tdbwkb> tableinfo animals
Retrieving structure of table animals.dat...
Table columns:
  1 S40  Name
  2 A    Area, Values = Land,Water,Air
  3 P    Image
  4 M    Description
  5 N    RecordId
Indexes:
animals.inr                   RecordId:4
animals.id                    Name:40

tdbwkb> altertable animals n2=S40Family
Restructuring table animals.dat to these columns:
  1 S40  Name
  2 S40  Family
  3 A    Area, Values = Land,Water,Air
  4 P    Image
  5 M    Description
  6 N    RecordId

tdbwkb> newindex animals byfamily Family,Name

tdbwkb> tableinfo animals
Retrieving structure of table animals.dat...
Table columns:
  1 S40  Name
  2 S40  Family
  3 A    Area, Values = Land,Water,Air
  4 P    Image
```

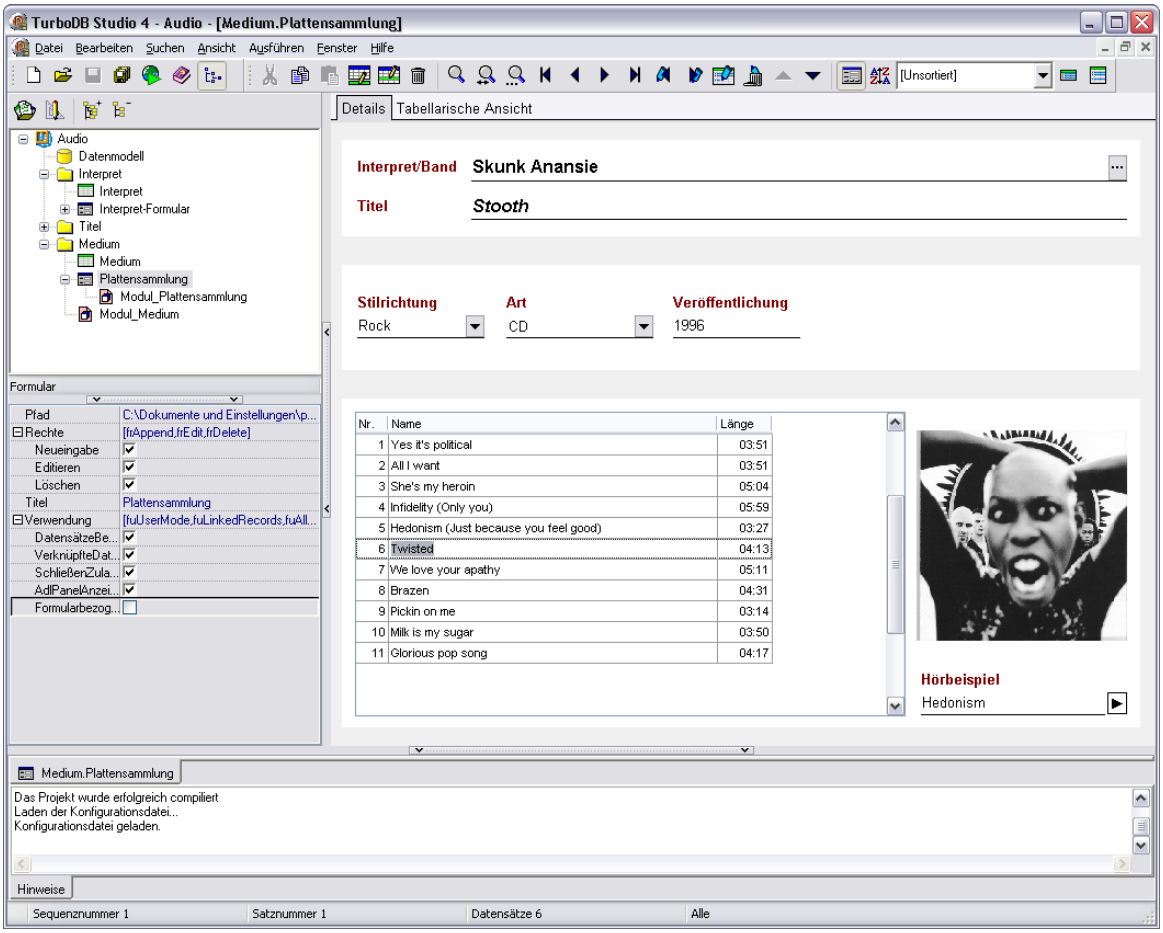
```
  5 M    Description
  6 N    RecordId
Indexes:
animals.inr                    RecordId:4
animals.id                     Name:40
byfamily.ind                   Family:40, Name:40
```

1.3.5.5 TurboDB Studio

This is our tool for creating Windows client applications for Turbo Database. Using TurboDB Studio you create forms and reports for interactive applications and printing. TurboDB Studio enables you to create customized executables with your own name, splash screen, menus and toolbars in a couple of hours. Users of TurboDB can manage their TurboDB database very quickly, enter or modify data and print all kind of reports.

You can use TurboDB Studio to

- Manage TurboDB database more comfortably then with TurboDB Viewer
- Prototype your TurboDB applications very rapidly
- Provide additional customized management tools to the users of TurboDB application (includes reporting)
- Create full-sized Windows applications based on TurboDB



More information on *TurboDB Studio* can be found on the the dataweb homepage (currently only in German).

1.3.5.6 TurboDB Data Exchange

Another text-based free-ware tool that reads and writes data from and to TurboDB tables. Formats supported by *tdbDataX* are Text, dBase, TurboDB, XML and ADO. You can download it from http://www.turbodb.de/.

Index

- C -

- D -

- E -

- F -

- G -

- H -

- I -

- J -

- K -

- L -

- M -

- N -

- O -

- P -

- Q -

- R -

- S -

- T -

- U -

- V -

- W -

- Y -