Accuracer Developer's Guide

Place your own product logo here and modify the layout of your print manual/PDF:
In Help & Manual, click "Tools" > "Print Manual Designer" and open this manual template to edit it.

Title page 1

Use this page to introduce the product

---

by

This is "Title Page 1" - you may use this page to introduce your product, show title, author, copyright, company logos, etc.

This page intentionally starts on an odd page, so that it is on the right half of an open book from the readers point of view. This is the reason why the previous page was blank (the previous page is the back side of the cover)

Accuracer Developer's Guide

Printed: August 2009 in (whereever you are located)

Publisher

...enter name...

Managing Editor

...enter name...

Technical Editors

...enter name...
...enter name...

Cover Designer

...enter name...

Team Coordinator

...enter name...

Production

...enter name...

Special thanks to:

All the people who contributed to this document, to mum and dad and grandpa, to my sisters and brothers and mothers in law, to our secretary Kathrin, to the graphic artist who created this great product logo on the cover page (sorry, don't remember your name at the moment but you did a great work), to the pizza service down the street (your daily Capricciosas saved our lives), to the copy shop where this document will be duplicated, and and and...

Last not least, we want to thank EC Software who wrote this great help tool called HELP & MANUAL which printed this document.

Table of Contents

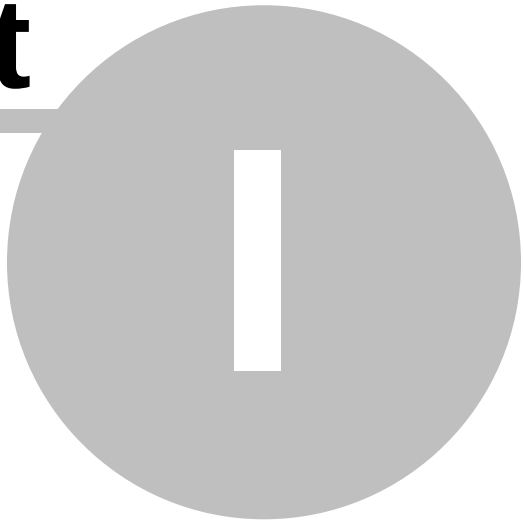
Foreword

This is just another title page
placed between table of contents
and topics

Top Level Intro

This page is printed before a new
top-level chapter starts

Part



1Tuesday, August 04, 2009 Accuracer Developer's Guide

1.1 Introduction

Accuracer: High-performance single file multi-user database with SQL for Delphi, C++ Builder, Kylix and ODBC

Accuracer is a fast single file multi-user and client-server database system with SQL support.

Main Features:

- Single file database
- Multi-User and Multi-Thread access
- Client-Server Engine 67
- No BDE; no DLLs;
- SQL'92 with subqueries and DDL statements
- Referential Integrity Support (SQL'99 compliance)
- Reverse engeneering (Export tables to SQL script)
- Full source code included
- Unmatched ease-of-use, lots of demos
- Small footprint
- 100% compatibility with standard DB-aware controls
- No Royalties
- BLOB and Varchar support (with optionally compression)
- Memory tables support (including SQL and DDL)
- BatchMove component
- Strong encryption of the database file (AES, Blowfish, Twofish, DES, etc.)
- Backup and Restore
- Triggers - database and server events
- ODBC driver available
- Windows/Linux platforms: Delphi, C++ Builder and Kylix versions available
- Transactions support, isolation level READ COMMITTED

Visit our web site to read latest news and view a detailed specification:
http://www.accuracer.com
or
http://www.aidaim.com

1.2 Features Overview

Functionality

- Single database file
- Multi-User and Multi-Thread 62 support
- Client-Server engine 67
- A subset of **SQL'92 including DDL** operators is supported by TACRQuery component. With Accuracer you can create SQL scripts 17 for creating tables, inserting, editing and deleting records, retrieving data by SELECT command. Read SQL Reference book in this Guide to learn more about SQL implemented in Accuracer

- Reverse engeneering (Export tables to SQL script)
- Advanced **search engine**. Accuracer supports 'LIKE' operator with wildcards '%' and '_', as well as 'IS NULL' and 'IS NOT NULL' in filters and queries.
- Full **multiple index** support, i.e. numerous fields in a table may comprise an index. Accuracer provides descending and ascending indexes, case-sensitive and insensitive indexes for string fields.
- Shareable in-memory tables. Accuracer supports simultaneous access to a table by multiple TAccuracer components within a single application.
- Minimum, Maximum and Default values support.
- Required fields support (NOT NULL constraint).
- Primary and unique indexes support.
- Autoincrement fields with lots of settings (minimum and maximum value, increment, cycled) based on Sequences
- Memory tables support with SQL & DDL
- BatchMove component
- RepairDatabase is implemented in TACRDatabase
- Executable database files support
- Backup and Restore support
- Triggers - database and server events
- Transactions support 65, READ COMMITTED isolation level.

Compactness

- Short compiled code with approximate size 740 Kb, no external drivers (such as BDE) required.
- Small memory consumption by Accuracer database engine.
- Varchar 14 and WideVarchar field types support with optionally compression 14.
- Fast BLOB data **compression**. Your large data fields will need less memory. Accuracer can compress data on the fly. The compression routines used in the Accuracer are much faster than most of popular archivers like PKZip, WinRar, Arj.
- CompactDatabase method in TACRDatabase allows to compact a database file
- Automatic reducing of the database file size in case of deleting data from the end of file.

High performance

- **Fast** search by B-tree indexes. At the moment Accuracer is one of the fastest existing single file databases for Delphi and C++ Builder.
- High-speed memory operations **performance** is achieved by means of using specially optimized memory manager and tuned algorithms.
- Quick operations with strings. Accuracer compares strings up to **3 times faster** than standard Delphi string routines. High performance is achieved by using a special library written in Assembler and an advanced sorting algorithm.

- Advanced **SQL optimizer** often makes query execution significantly faster by choosing the best execution plan.

Compatibility

- Accuracer supports most of TTable field data types, including BLOB fields, moreover it allows to create string and wide **string fields of any fixed and variable length**.
- Accuracer is fully compatible with **standard DB-aware** visual controls such as QuickReport, DBGrid, DBNavigator, DBImage, DBMemo, DBRichEdit, as well as with third party vendor's products supporting TDataset descendant components - FastReport, DBFlyTreeView and others.
- Calculated and lookup fields can be used in the same way as TTable.
- Most of TTable functions are supported including **Key** and **Range** methods.

Convenience

- Table **restructuring** is being performed in the easiest way keeping all the existing data.
- Data importing from and exporting to any dataset is supported. Accuracer provides you with the simplest way to import and export tables using ImportTable and ExportTable methods.
- Internationalization / localization support. All text search and sorting functions use current system locale, so localizing your program with Accuracer is a very simple task.
- **Unicode** support. All the text operations work with multi-byte encoding using ftWideString.
- Comprehensive help. Accuracer comes with full documentation presented in Accuracer Developer's Guide and Accuracer Reference.
- Lots of demos for different IDE - Delphi, C++ Builder, Kylix and ODBC.

Security

- Database encryption by best symmetric ciphers (AES, Blowfish, Twofish, DES, etc.)
- Direct setting of the encryption parameters - cipher mode (CTS, OFB, CBC, CFB), initial vector, binary key
- String passwords supported (RipeMD128 / RipeMD256 hash used)
- All pages inside the database files are encrypted, including all internal data like indexes, maps, directory
- New pages are filled with random data by default
- Secure random number generator based on LFSR algorithm
- Open Source encryption algorithms implementation (DEC 1 library by Hagen Reddmann)

Cross platform product

- VCL - Delphi 4,5,6,7, 2005, 2006, 2007 and C++ Builder 4,5,6, 2006
- CLX - Kylix 3 Delphi
- ODBC

1.3 Getting Help from Technical Support

Free Tech Support
Should you have any questions, comments or ideas on adding new possibilities and/or changing the product's functions, contact us at support@aidaim.com easily.

We consider any ideas and we may take them into account while creating new versions of our products.

If you encountered a problem, please, inform us about the following:

- Product name and version
- Compiler information: Delphi or C++ Builder, Version, Edition, Service Pack
- Environmental information: your OS and Service Pack
- Description of your problem (as much information as possible to retrieve the problem).
- Attach a test project where the problem could be reproduced (it helps us to solve your issue as soon as possible)

Typically AidAim Software Support Team answer messages in 24 hours, but depending on singularity and difficulty of your question it may take a bit longer.

1.4 How to Buy

To place an order and to get pricing information, visit our site www.aidaim.com.

Feel free to contact us at support@aidaim.com if you have any technical questions.

For Sales-related questions contact Sales Department at sales@aidaim.com.

If you are an established distributor or reseller and you wish to have AidAim products in your portfolio, please don't hesitate to contact us at sales@aidaim.com.

1.5 Working with Tables

1.5.1 Creating a database file

You can create a database file using one of these methods:
1) Run ACRManager utility and choose New Database item of the Database menu. Follow the instructions.
2) Use CreateDatabase method of TACRDatabase component. See CreateDatabase demo.

1.5.2 Setting up a table and database components

The following steps are general instructions for setting up a table component at design time. There may be additional steps you need to tailor a table's properties to the requirements of your application. If you need in-memory table you should not create a database component, just set InMemory property to true.

To create a database component,

1. Place TACRDatabase component from the Accuracer page of the Component palette in a data module or on a form, and set its DatabaseName property to a unique value appropriate to your application.
2. Set DatabaseFileName property to the path to the database file. You can use OpenDialog if you double click on this property in Object Inspector.

To create a table component,

1. If you need an in-memory table set InMemory property to True, otherwise set DatabaseName property to specify which TACRDatabase component will be used for connecting to the database file.
2. Place TACRTable component from the Accuracer page of the Component palette in a data module or on a form, and set its Name property to a unique value appropriate to your application.
3. Set the TableName property to the name of the table in the database.

To access the data encapsulated by a table component,

1. Place a data source component from the Data Access page of the Component palette in the data module or form, and set its DataSet property to the name of the table component.
2. Place a data-aware control, such as TDBGrid, on a form, and set the control's DataSource property to the name of the data source component placed in the previous step.
3. Set Connected property of TACRDatabase component to True if you use table from the database file. Skip this step if you use in-memory table.
4. Set the Active property of the table component to True.

1.5.3 Creating a table

Introduction

Creating tables is accomplished through the CreateTable method of the TAccuracer component. The properties used by the CreateTable method include the FieldDefs, IndexDefs, TableName and Exists properties.

Specifying the Fields to Create

The FieldDefs property is used to specify which fields to define for the new table. The FieldDefs property is an array of TFieldDef objects, each of which contains information about the field to create. You may add new TFieldDef objects using the Add method of the TFieldDefs object stored in the FieldDefs property. The Add method accepts the following parameters for the field being defined:

Field Name (String)	Field Name parameter indicates the name to give the field.
Data Type (TFieldType)	DataType parameter indicates the data type of the field Available TFieldType data types 74
Size (Word)	Size parameter indicates the size of the field. This should be specified for the String type only. For all other data types this parameter should be set as 0. For the String type this parameter indicates the length of the field. For the WideString type this

	parameter indicates the size of the wide string in bytes.
Required (Boolean)	Required parameter indicates whether or not the new field should be required (not Null) while adding or modifying records.

Advanced FieldDefs

Accuracer provides some advanced functionality that cannot be specified by FieldDefs like minimum, maximum and default values, autoincrement fields settings, blob and varchar fields compression settings. Use AdvFieldDefs instead of FieldDefs for specifing fields with advanced parameters. If you use AdvFieldDefs you should empty FieldDefs list by calling FieldDefs.Clear and vice versa. See Reference Guide for more information about AdvFieldDefs (TACRAdvFieldDef class).

Specifying the Indexes to Create

The IndexDefs property is used to specify which indexes to be defined for the new table. The IndexDefs property is an array of TIndexDef objects, each of them containing information about the index to create. You may add new TIndexDef objects using the Add method of the TIndexDefs object contained in the IndexDefs property. The Add method accepts the following parameters for the index being defined:

Index Name (String)	Index Name parameter contains the name to be given to the index.
Fields List (String)	Fields List parameter contains the list of fields to be included into the index. Multiple field names specified in this parameter should be separated with a semicolon (;).
Index Options (TIndexOptions)	Index Options parameter provides information about the type of index being created (please see the component reference supplied with Accuracer for more information on the available TIndexOption options).

Setting the Table Information

The TableName property specifies the name for the created table.

Creating the Table

Any table can have a primary key on fields of any type. You may create this key with the index by means [ixPrimary] option.
The final step in creating a table is to call the CreateTable method. It is also recommended to check the Exists property of the TAccuracer component first to make sure that you are not attempting to overwrite an existing table. The following example shows how to create the CUSTOMER table included with the Delphi demo in existing database DBDemos.adb' using the CreateTable method:

```
begin
  with MyAccuracer do
    begin
    TableName:='customer';
    with FieldDefs do
      begin
      Clear;
```

```
      Add('CustNo',ftAutoInc,0,False);
      Add('Company',ftString,30,False);
      Add('Addr1',ftString,30,False);
      Add('Addr2',ftString,30,False);
      Add('City',ftString,15,False);
      Add('State',ftString,20,False);
      Add('Zip',ftString,10,False);
      Add('Country',ftString,20,False);
      Add('Phone',ftString,15,False);
      Add('FAX',ftString,15,False);
      Add('TaxRate',ftFloat,0,False);
      Add('Contact',ftString,20,False);
      Add('LastInvoiceDate',ftDateTime,0,False);
      end;
    with IndexDefs do
      begin
      Clear;
      Add('PrimaryKey','CustNo',[ixPrimary]);
      Add('ByCompany','Company',[ixCaseInsensitive]);
      end;
    if not Exists then
      CreateTable;
    end;
end;
```

1.5.4 Opening and closing a table

Introduction

After setting up Accuracer component 6 and creating a table 6, just open the table to view and edit table's data in a data-aware control such as TDBGrid. There are two ways to open a table. You can set its Active property to True, or you can call its Open method. The following example shows how to use the Open method to open a table called 'customers' with a TAccuracer component called MyAccuracer:

```
begin
  // table settings
  with MyAccuracer do
    begin
     TableName:='customers';
     ReadOnly:=False;
     Open;
    end;
end;
```

The ReadOnly property causes the current table with the name in the TableName property to be opened read-only, which means that the current application will be unable to modify the contents of the table until the table is closed and re-opened with write access (ReadOnly=False).

There are two ways to close a table. You can set its Active property to False, or you can call its Close method. Active controls associated with the table's data source are cleared.
The following example shows how to use the Close method to close a table with a TAccuracer component called MyAccuracer:

```
begin
```

```
  MyAccuracer.Close;
end;
```

Important Note:
In-memory table is not deleted after closing table. You should call Table.DeleteTable for physical remove of the table data.

1.5.5 Navigating tables

Introduction

There are the following basic methods you can use in application code to move to different records:

Method	Description
First	Moves to the first row of the table.
Last	Moves to the last row of the table.
Next	Moves to the next row of the table.
Prior	Moves to the previous row of the table.
SetRecNo	Moves a cursor to the specified record (RecNo is calculated with all filters applied to the table)

Note
All these methods are based upon the current index order.

In addition to these methods, the following table describes two Boolean properties of tables that provide useful information when iterating through the records in a table.
The BOF and EOF properties indicate whether the record pointer is at the beginning of the table or it is at the end of the table, respectively.

The following code illustrates one of the ways you might code a record-processing loop for an Accuracer component called CustTable:

```
CustTable.First;  Go to first record, which sets EOF False
while not CustTable.EOF do  Cycle until EOF is True
begin
  Process each record here
 ...
 CustTable.Next;  EOF False on success; EOF True when Next fails on last record
end;
```

1.5.6 Filtering records

Introduction

Setting filters on tables is accomplished through several methods of the TAccuracer component. The basic filter properties include the Filter, FilterOptions and Filtered properties. The OnFilterRecord event is used to implement a callback filter event that can be used to filter records using Delphi code. All filter operations keep current index order.

Setting the Filter property

To create a filter using the *Filter* property, set the value of the property to a string that contains the filter conditions. The string contains the filter's test condition. For example, the following statement creates a filter that tests a table's State field to see if it contains a value for the state of California:

```
table1.Filter := 'State = ' + QuotedStr('CA');
```

you can also supply a value for *Filter* based on the text entered in a control. For example, the following statement assigns the text from an edit box to *Filter*:

```
table1.Filter := Edit1.Text;
```

you can also create a condition for boolean fields:

```
table1.Filter := 'Married = TRUE';
```

you can also create a string based both on hard-coded text and on data entered by a user in a control:

```
table1.Filter := 'State = ' + QuotedStr(Edit1.Text);
```

you can also compare field values to literals, and to constants using the following logical and comparison operators:

Operator	Meaning
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to
AND	Tested statements are both True
NOT	Tested statement is not True
OR	At least one of two statements tested is True
[NOT] LIKE	Extended operator for string field value comparisons with wildcards %,
IS [NOT] NULL	Extended operator for determining whether a field value is NULL

Using combinations of the above listed operators you can create fairly sophisticated filters. For example, the following statement checks if the two test conditions meet when searching for a record:

```
(Custno > 1400) AND (Custno < 1500);
```

Setting filter options

The FilterOptions property enables you to specify whether or not a filter that compares string-based fields accepts records based on partial comparisons and whether or not string comparisons are case-sensitive. FilterOptions is a set property that can be an empty set (the default), or that can contain either or both of the following values:

Value	Meaning
foCaseInsensitive	Ignore case when comparing strings.

foPartialCompare	Disable partial string matching (i.e., do not match strings ending with an asterisk (*)).

For example, the following statements set up a filter that ignores case when comparing values in the State field:

```
FilterOptions := [foCaseInsensitive];

Filter := '"State" = "CA"';
```

Activating filter

Set the Filtered property to True.
When filtering is enabled, only those records that meet the filter criteria are available to an application. Filtering is always a temporary condition. You can turn off filtering by setting the Filtered property to False.

1.5.7 Searching records

Introduction

Searching records on tables is accomplished through several methods of the TAccuracer component. The basic search methods are FindFirst, FindLast, FindNext, and FindPrior.
You may also use Locate and Lookup methods to find one matched record.
All search operations use current index order.
Search also performs in a master/detail and filtered tables.

Searching by Find methods.

Searching records with an Accuracer component by Find methods is a three-step process:

1. Set filter.property 9
2. Set filter options for string-based filter tests, if necessary.
3. Call some of the following navigational methods: FindFirst(), FindLast(), FindNext(), and FindPrior().

All these navigational methods position the record pointer to a matching record (if any), make it current, and return True. If a matching record is not found, the record pointer position is unchanged (remains as it is), and these methods return False.You can check the status of the *Found* property to wrap these calls, and only take action when *Found* is True. For example, if the record pointer is already on the last matching record in the table, and you call FindNext, the method returns False, and the current record remains unchanged.

Using Locate

Locate moves the cursor to the first row matching a specified set of search criteria. In its simplest form, you pass Locate the name of a column to search, a field value to match, and an options flag specifying whether the search is case-insensitive or if it can use partial-key matching. For example, the following code moves the cursor to the first row in the CustTable where the value in the Company column is "Professional Divers, Ltd.":

```
var
  LocateSuccess: Boolean;
  SearchOptions: TLocateOptions;
```

```
begin
  SearchOptions := [loPartialKey];
  LocateSuccess := CustTable.Locate('Company', 'Professional Divers, Ltd.',    SearchOptions);
end;
```

If Locate finds a match, the first record containing the match becomes the current record. Locate returns True if it finds a matching record, False if it does not. If a search fails, the current record does not change.
The real power of Locate comes into play when you want to search on multiple columns and specify multiple values to search for. Search values are variants, which enables you to specify different data types in your search criteria. To specify multiple columns in a search string, separate individual items in the string with semicolons.

Because search values are variants, if you pass multiple values, you must either pass a variant array type as an argument (for example, the return values from the Lookup method), or you must construct the variant array on the fly using the VarArrayOf function. The following code illustrates a search on multiple columns using multiple search values and partial-key matching:

```
with CustTable do
  Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver','P']), loPartialKey);
```

Locate uses the fastest possible method to locate the matching record.

Using Lookup

Lookup searches for the first row that matches specified search criteria. If it finds a matching row, it forces the recalculation of any calculated fields and lookup fields associated with the dataset, then returns one or more fields from the matching row. Lookup does not move the cursor to the matching row; it only returns values from it.
In its simplest form, you pass Lookup the name of field to search, the field value to match, and the field or fields to return. For example, the following code looks for the first record in the CustTable where the value of the Company field is "Professional Divers, Ltd.", and returns the company name, a contact person, and a phone number for the company:

```
var
  LookupResults: Variant;
begin
with CustTable do
  LookupResults := Lookup('Company', 'Professional Divers, Ltd.', 'Company; Contact; Phone');
end;
```

Lookup returns values for the specified fields from the first matching record it finds. Values are returned as variants. If more than one return value is requested, Lookup returns a variant array. If there are no matching records, Lookup returns a Null variant. For more information about variant arrays, see the online help.
The real power of Lookup comes into play when you want to search on multiple columns and specify multiple values to search for. To specify strings containing multiple columns or result fields, separate individual fields in the string items with semi-colons.

Because search values are variants, if you pass multiple values, you must either pass a variant array type as an argument (for example, the return values from the Lookup method), or you must construct the variant array on the fly using the VarArrayOf function. The following code illustrates a lookup search on multiple columns:

```
var
  LookupResults: Variant;
begin
```

```
with CustTable do
  LookupResults := Lookup('Company; City', VarArrayOf(['Sight Diver', 'Christiansted']),
'Company; Addr1; Addr2; State; Zip');
end;
```

Lookup also uses the fastest possible method to locate the matching record.

1.5.8 Sorting records

Introduction

You may use the IndexName and IndexFieldNames properties to set the current index order, and consequently, sort the current table based upon the index definition for the selected index order. The IndexName property is used to set the name of the current index. If this property is set to blank ('') records are not sorted and shown in physical order. The following example shows how you would set the current index order to an index called CustomerName:

```
begin
with MyAccuracer do
     begin
      IndexName:='CustomerName';
       do something
     end;
end;
```

Please note that changing the index order can cause the current record pointer to move to a different position in the table (but not necessarily move off of the current record unless the record has been changed or deleted by another user), so please be sure to call the *First* method after setting the *IndexName* property if you want to have the record pointer set to the beginning of the table based upon the next index order. Since the logical record numbers are based upon the index order the record number may also change. If you attempt to set the *IndexName* property to a non-existent index an exception will be raised.
The *IndexFieldNames* property is used to set the current index order by specifying the field names of the desired index instead of the index name. Multiple field names should be separated with a semicolon. Using the *IndexFieldNames* property is desirable in cases where you are trying to set the current index order based upon a known set of fields and do not have any knowledge of the index names available. The *IndexFieldNames* property will attempt to match the given number of fields with the same number of beginning fields in any of the available primary or secondary indexes. The following example shows how you would set the current index order to a secondary index called *CustomerName* that consists of the *CustomerName* field and the *CustomerNo* field:

```
begin
  with MyAccuracer do
     begin
      IndexFieldNames:='CustomerName;CustomerNo';
       do something
     end;
end;
```

Please note that if Accuracer cannot find any indexes that match the desired field names an exception will be raised. If you are using this method of setting the current index order you should also be prepared to trap for this exception and deal with it appropriately.

1.5.9 BLOB fields use

Introduction

Use of BLOB fields in Accuracer is the same as in TTable component.
BLOB fields compression is transparent, so you can easily use it if you store large amounts of data in BLOB fields.

Usage
Use TACRBlobStream to access or modify the value of a BLOB field in an Accuracer. TACRBlobStream is a stream object that provides services allowing applications to read from or write to field objects that represent Binary large object (BLOB) fields.

TACRBlobStream allows objects that have no specialized knowledge of how data is stored in a BLOB field to read or write such data by employing the uniform stream mechanism.

To use a BLOB stream, create an instance of TACRBlobStream, use the methods of the stream to read or write the data, and then close the BLOB stream. Do not use the same instance of TACRBlobStream to access data from more than one record. Instead, create a new TACRBlobStream object every time you need to read or write BLOB data on a new record.

Example

The following example reads the data from a memo field into a blob stream and displays it in a memo control.

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Buffer: PChar;
  MemSize: Integer;
  Stream: TACRBlobStream;
begin
  Stream := TACRBlobStream.Create(MyAccuracer.FieldByName('Notes') as TBlobField,
bmRead);
  try
    MemSize := Stream.Size;
    Inc(MemSize); Make room for the buffer's null terminator.
    Buffer := AllocMem(MemSize);     Allocate the memory.
    try
      Stream.Read(Buffer^, MemSize); Read Notes field into buffer.
      Memo1.SetTextBuf(Buffer);      Display the buffer's contents.
    finally
      FreeMem(Buffer, MemSize);
    end;
  finally
    Stream.Free;
  end;
end;
```

1.5.10 BLOB and Varchar fields

Introduction

Accuracer supports string fields of variable length - varchar SQL field type or ftString field type. Varchar fields allows to get more compact database file, as they stores only actually number of

characters for each field of each record, while fixed length character fields always stores maximum number of characters. Aslo varchar fields like a BLOB fields can be optionally compressed.
We will use SQL terminology for field types naming to avoid misunderstanding.
Char fields is a fixed length character fields, while Varchar is a variable length character fields.

BLOB and Varchar fields compression is transparent, so you can easily use it if you store large amounts of data.

Usage

You can specify a compression settings for BLOB or Varchar fields:
1) Using ACRManager utility
2) Using AdvFieldDefs property of TACRTable - see Reference Guide, TACRAdvFieldDef class.
3) Using SQL script for creating tables - see CREATE TABLE statement 56 topic

How to choose a compression settings

First you should choose one of three compression algorithms: ZLIB, BZIP or PPM. ZLIB is the fastest, but provides lower compression rate. However it is recommended when you need to store short data (~ 1 Kb or less). BZIP is a little slower, but usually provides better compression than ZLIB. Both of these algorithms decompresses data (read operations) much faster than compresses data (write operations). The speed difference may be up to 10 times.
PPM is a much slower algorithm, providing best compression rate on most types of data. PPM decompresses data a bit slower than compresses it. This algorithm is recommended for large amounts of data (100 Kb or more).

After that you should choose a proper compression mode - integer value from 1 (minimum compression rate and maximum speed) to 9 (maximum compression rate and slower speed).

BLOB block size parameter is used only for BLOB (or Memo) fields, but not for Varchar fields. This is a size of memory buffer required for streaming compression. Default value is 100 Kb. If you set smaller value the memory buffer will be smaller, but it can lead to lower compression rate and speed. Larger value will provide better compression rate (especially for PPM).

Note: Memory usage is determined by compression algoirhm and mode. Large mode require large amount of memory in BZIP and PPM. PPM requires from 2 Mb to 100 Mb of RAM to operate, BZIP requires from 100 Kb to 900 Kb of RAM and ZLIB requires up to 256 Kb of RAM. If you use BLOB (or Memo) fields Accuracer allocates a memory buffer which size is specified in BLOB block size parameter (100 Kb by default). Varchar fields does not allocates this buffer.
Note: If you specify a compression algorithm None the compression mode value is ignored, while BLOB block size is still active for BLOB fields.

How to choose between Varchar, Char, Memo and BLOB field types

If you need to store binary values you should use BLOB fields.
As for text data it generally depends on the operations you need to perform against this data and on size of the data.
If need any kind of searching or filters on this text you should use either Varchar or Char field types. Search in Memo fields is not supported (it can be done only by OnFilterRecord event, but it will operate rather slow).
If you do not need searching and your data will be rather large you should use Memo fields.

Varchar fields works faster than memo fields and usually achieve better compression rate.
Char fields provides better performance than Varchar fields, but requires more space, especially when there are many empty or short values. Varchar fields requires from 6 to 30 bytes per value for storing necessary link information and headers. Null values takes only 6 bytes reserved for link

in the record, not null values will take additionally 20-24 bytes for headers depending on compressed data size is smaller than page size (24 bytes) or not (20 bytes plus size of empty space at last page).

1.6 Advanced Operations with Tables

1.6.1 Restructuring a table

Introduction

Restructuring tables is executed by means of the RestructureTable method of the TACRTable component. The properties used by the RestructureTable method include the RestructureFieldDefs and RestructureIndexDefs properties.

Specifying the New Fields Structure

The RestructureFieldDefs property is used to specify which fields to define for the restructured table. The RestructureFieldDefs property is an array of TFieldDef objects, each of which contains information about the field to create. When table is open RestructureFieldDefs property contains field definitions of all existing fields. So you should not define all fields. You may add, modify or delete some fields definitions only. You may add new TFieldDef objects using the Add method of the TFieldDefs object stored in the FieldDefs property. The Add method accepts the following parameters for the field being defined:

Field Name (String)	Field Name parameter indicates the name to be given to the field.
Data Type (TFieldType)	DataType parameter indicates the data type of the field Available TFieldType data types: ftInteger, ftSmallInt, ftFloat, ftDateTime, ftBLOB, ftString (any fixed length string)
Size (Word)	Size parameter indicates the size of the field. This should be specified for the String type only. For all other data types this parameter should be set as 0. For the String type this parameter indicates the length of the field.
Required (Boolean)	Required parameter indicates whether or not the new field should be required (not Null) while adding or modifying records.

Specifying the Indexes in a Restructured Table

The RestructureIndexDefs property is used to specify which indexes to be defined for the restructured table. The RestructureIndexDefs property is an array of TIndexDef objects, each of them containing information about the index to create. When table is open RestructureIndexDefs property contains index definitions for current table. So you should not define all indexes. You may add, modify or delete some index definitions only. You may add new TIndexDef objects using the Add method of the TIndexDefs object contained in the IndexDefs property. The Add method accepts the following parameters for the index being defined:

Index Name (String)	Index Name parameter contains the name to be given to the index.
Fields List (String)	Fields List parameter contains the list of fields to be included into the index. Multiple field names specified in this parameter should be separated with a semicolon (;).
Index Options (TIndexOptions)	Index Options parameter provides information about the type of index being created (please see the component reference supplied

with Accuracer for more information on the available TIndexOption options).

Restructuring the Table

The final step in restructuring a table is to call the RestructureTable method. The following example shows how to restructure the CUSTOMER.DAT table included with the Delphi demo data using the RestructureTable method:
(Structure of this table is described in example from Creating a table topic)

```
MyAccuracer.Open;
MyAccuracer.Close;
 with MyAccuracer do
  begin
   modify fields structure
   with RestructureFieldDefs do
    begin
     // add new field
     Add('Customer Name',aftString,300,False);
     // set new length for Company field
     Find('Company').Size := 100;
     // delete 'Birthday' field
     DeleteFieldDef('Birthday');
    end;
   modify index definitions
   with RestructureIndexDefs do
    begin
     // add new index for Customer Name field
     Add('CustomerName_Index','Customer Name',[ixCaseInsensitive]);
     // update primary index
     Find('PrimaryKey').Fields := 'Customer ID';
    end;
   // change only fields structure, don't modify other parameters such as encryption
   RestructureTable;
  end;
MyAccuracer.Open;
```

1.7 SQL Reference

1.7.1 Overview

Introduction

Accuracer supports a subset of SQL'92 commands. It includes most widely used SQL statements for data manipulation and definition - SELECT, INSERT, UPDATE, DELETE, CREATE TABLE, ALTER TABLE, DROP TABLE, CREATE INDEX, DROP INDEX. Accuracer contains native TACRQuery component that allows to run SQL commands and scripts in fast and easy way. Accuracer does not use any third party components or drivers. This approach allows to achieve high performance on most of SQL queries and makes distribution of applications very easy. No BDE, no dlls, no drivers needed.

For accessing in-memory tables specify MEMORY option before table name in any SQL statement.

You can run SQL Scripts by TACRQuery component - just assign script text to SQL property of TACRQuery.
SQL commands should be separated by semi-colon (**;**).

Supported SQL commands:

- SELECT 48 Statement
- INSERT 53 Statement
- UPDATE 53 Statement
- DELETE 54 Statement
- CREATE DATABASE 55 Statement
- DROP DATABASE 55 Statement
- CREATE TABLE 56 Statement
- ALTER TABLE 58 Statement
- DROP TABLE 60 Statement
- CREATE INDEX 60 Statement
- DROP INDEX 61 Statement
- START TRANSACTION 61 Statement
- COMMIT 62 Statement
- ROLLBACK 62 Statement

See also:
Naming conventions 18
Operators 25
Functions 28

1.7.2 Naming conventions and reserved words

Introduction

SQL of Accuracer supports the most flexible way to name databases, tables and columns.

Table Name
Table name could be a single word or a multiple words. Multiple words names or names matches any reserved word must be enclosed by back, single or double quotes, or square brackets. It is recommended to use square brackets or back quotes, as it prevents from parsing names as string constants.

For example:

```
SELECT  *
FROM  [Detail Parts]

SELECT  *
FROM  `Detail Parts`
```

You can also use correlation name:

```
SELECT  DP.PartNo
FROM  [Detail Parts] DP
```

Parameter names

Parameter names starts with colon symbol.

Parameter names with delimiter characters or reserved words must be quoted by quotes, double quotes or back quotes:

```
INSERT INTO test1 VALUES (:test2);
INSERT INTO test1 VALUES (:`test 2`);
INSERT INTO test1 VALUES (:"test 2");
INSERT INTO test1 VALUES (:'test 2');
```

Column Name

Column name could be a single word or a multiple words. Multiple words names or names matches any reserved word must be enclosed by back, single or double quotes, or square brackets. It is recommended to use square brackets or back quotes, as it prevents from parsing names as string constants.

For example:

```
SELECT  Orders.[Cust No]
FROM  Orders

SELECT  Orders.`Cust No`
FROM  Orders
```

You can also use short correlation name for columns:

```
SELECT  C.Name AS CustName
FROM  Customer C
WHERE CustName LIKE 'Bill%'
```

Comments

You can use comments in SQL quieries text to keep remarks or some useful information about the query.
Single-line comments should be started with '--' symbols:

```
-- This is a single-line comment
SELECT  *  FROM  CUSTOMERS
```

Another variant of comments is enclosing text into /* and */ symbols.
It can be used for temporarily removing of some query parts:

```
SELECT  *  FROM  CUSTOMERS
/* WHERE (Name = 'Mike') */
ORDER BY CustNo
```

Reserved Words

Here is the list of words reserved by Accuracer's SQL Engine. Some of them are not really supported, but reserved for further implementations.

```
  'ABSOLUTE'
  ,'ACTION'
  ,'ADD'
  ,'ALL'
  ,'ALLOCATE'
  ,'ALTER'
  ,'AND'
  ,'ANY'
  ,'ARE'
  ,'AS'
```

```
,'ASC'
,'ASSERTION'
,'AT'
,'AUTHORIZATION'
,'AVG'
,'BEGIN'
,'BETWEEN'
,'BIT'
,'BIT_LENGTH'
,'BOTH'
,'BY'
,'CASCADE'
,'CASCADED'
,'CASE'
,'CAST'
,'CATALOG'
,'CHAR'
,'CHARACTER'
,'CHAR_LENGTH'
,'CHARACTER_LENGTH'
,'CHECK'
,'CLOSE'
,'COALESCE'
,'COLLATE'
,'COLLATION'
,'COLUMN'
,'COMMIT'
,'CONNECT'
,'CONNECTION'
,'CONSTRAINT'
,'CONSTRAINTS'
,'CONTINUE'
,'CONVERT'
,'CORRESPONDING'
,'COUNT'
,'CREATE'
,'CROSS'
,'CURRENT'
,'CURRENT_DATE'
,'CURRENT_TIME'
,'CURRENT_TIMESTAMP'
,'CURRENT_USER'
,'CURSOR'
,'DATE'
,'DAY'
,'DEALLOCATE'
,'DEC'
,'DECIMAL'
,'DECLARE'
,'DEFAULT'
,'DEFERRABLE'
,'DEFERRED'
,'DELETE'
,'DESC'
,'DESCRIBE'
,'DESCRIPTOR'
,'DIAGNOSTICS'
```

```
,'DISCONNECT'
,'DISTINCT'
,'DOMAIN'
,'DOUBLE'
,'DROP'
,'ELSE'
,'END'
,'END-EXEC'
,'ESCAPE'
,'EXCEPT'
,'EXCEPTION'
,'EXEC'
,'EXECUTE'
,'EXISTS'
,'EXTERNAL'
,'EXTRACT'
,'FALSE'
,'FETCH'
,'FIRST'
,'FLOAT'
,'FOR'
,'FOREIGN'
,'FOUND'
,'FROM'
,'FULL'
,'GET'
,'GLOBAL'
,'GO'
,'GOTO'
,'GRANT'
,'GROUP'
,'HEX'
,'HAVING'
,'HOUR'
,'IDENTITY'
,'IF'
,'IMMEDIATE'
,'IN'
,'INDICATOR'
,'INITIALLY'
,'INNER'
,'INPUT'
,'INSENSITIVE'
,'INSERT'
,'INT'
,'INTEGER'
,'INTERSECT'
,'INTERVAL'
,'INTO'
,'IS'
,'ISNULL'
,'ISOLATION'
,'JOIN'
,'KEY'
,'LANGUAGE'
,'LAST'
,'LEADING'
```

```
,'LEFT'
,'LEVEL'
,'LIKE'
,'LOCAL'
,'LOWER'
,'MATCH'
,'MAX'
,'MEMORY'
,'MIME64'
,'MIN'
,'MINUS'
,'MINUTE'
,'MODULE'
,'MONTH'
,'NAMES'
,'NATIONAL'
,'NATURAL'
,'NCHAR'
,'NEXT'
,'NO'
,'NOFLUSH'
,'NOT'
,'NULL'
,'NULLIF'
,'NUMERIC'
,'OCTET_LENGTH'
,'OF'
,'ON'
,'ONLY'
,'OPEN'
,'OPTION'
,'OR'
,'ORDER'
,'OUTER'
,'OUTPUT'
,'OVERLAPS'
,'PAD'
,'PARTIAL'
,'POSITION'
,'PRECISION'
,'PREPARE'
,'PRESERVE'
,'PRIMARY'
,'PRIOR'
,'PRIVILEGES'
,'PROCEDURE'
,'PUBLIC'
,'READ'
,'REAL'
,'REFERENCES'
,'RELATIVE'
,'RESTRICT'
,'REVOKE'
,'RIGHT'
,'ROLLBACK'
,'ROWS'
,'SCHEMA'
```

```
,'SCROLL'
,'SECOND'
,'SECTION'
,'SELECT'
,'SESSION'
,'SESSION_USER'
,'SET'
,'SIZE'
,'SMALLINT'
,'SOME'
,'SPACE'
,'SQL'
,'SQLCODE'
,'SQLERROR'
,'SQLSTATE'
,'START'
,'SUBSTRING'
,'SUM'
,'SYSTEM_USER'
,'TABLE'
,'TEMPORARY'
,'THEN'
,'TIME'
,'TIMESTAMP'
,'TIMEZONE_HOUR'
,'TIMEZONE_MINUTE'
,'TO'
,'TOP'
,'TRAILING'
,'TRANSACTION'
,'TRANSLATE'
,'TRANSLATION'
,'TRIM'
,'TRUE'
,'UNION'
,'UNIQUE'
,'UNKNOWN'
,'UPDATE'
,'UPPER'
,'USAGE'
,'USER'
,'USING'
,'VALUE'
,'VALUES'
,'VARCHAR'
,'VARYING'
,'VIEW'
,'WHEN'
,'WHENEVER'
,'WHERE'
,'WITH'
,'WORK'
,'WRITE'
,'YEAR'
,'ZONE'
,'PASSWORD'             // for DDL commands
,'BLOBBLOCKSIZE'        // for DDL commands
```

```
,'BLOBCOMPRESSIONMODE'   // for DDL commands
,'BLOBCOMPRESSIONALGORITHM'// for DDL commands
,'LASTAUTOINC'           // for DDL commands
,'MODIFY'              // alter table blablabla modify ...
,'NEW'                 // for NEW PASSWORD in ALTER TABLE
,'INDEX'               // for CREATE INDEX ...
,'NOCASE'              // for CREATE INDEX ... NOCASE ..
,'LTRIM'
,'RTRIM'
,'POS'
,'LENGTH'
,'SYSDATE'             // DateTime function
,'NOW'
,'TOBLOB'              // TOBLOB function
,'TODATE'              // TODATE function
,'TOSTRING'            // TOSTRING function
,'AUTOINDEXES'         // for DDL AutoIndexes
,'NOAUTOINDEXES'       // for DDL AutoIndexes
,'INCREMENT'
,'LASTVALUE'
,'MAXVALUE'
,'MINVALUE'
,'CYCLED'
,'NOMAXVALUE'
,'NOMINVALUE'
,'NOCYCLED'
,'INITIALVALUE'
,'RENAME'
,'QUARTER'
,'WEEKDAY'
,'DAYOFWEEK'
,'DAYNAME'
,'MONTHNAME'
,'MSECOND'
,'ABS'
,'CEILING'
,'CEIL'
,'FLOOR'
,'MOD'
,'POWER'
,'POW'
,'RANDOM'
,'RAND'
,'ROUND'
,'SIGN'
,'TRUNCATE'
,'TRUNC'
,'SHL'
,'SHR'
,'DATABASE'
,'FILE'
,'PAGESIZE'
,'MAXSESSIONSCOUNT'
,'CUMSUM'
,'CUMPROD'
,'GROUP_CONCAT'
```

1.7.3 Using parameters

Parameters can be used for replacing data values in SQL statements. Parameters are identified by a preceding colon (:).
Parameter names with delimiter characters or reserved words must be quoted by quotes, double quotes or back quotes:

```
INSERT INTO test1 VALUES (:test2);
INSERT INTO test1 VALUES (:`test 2`);
INSERT INTO test1 VALUES (:"test 2");
INSERT INTO test1 VALUES (:'test 2');
```

You can use parameters with SELECT, INSERT, UPDATE or DELETE statements. Here are some examples how you can use parameters:

```
With ACRQuery1 do
 begin
  SQL.Clear;
  SQL.Add('INSERT INTO customer_Sort (Company,Address,CustNo)');
  SQL.Add('VALUES (:Company, :Address, :CustNo)');
  Params[0].AsString := 'AidAim Software';
  Params[1].AsString := 'US';
  Params[2].AsInteger := 4;
  ExecSQL;
 end;

With ACRQuery1 do
 begin
  SQL.Clear;
  SQL.Add('SELECT * FROM orders WHERE PaymentMethod = :PayMethod ');
  ParamByName('PayMethod').AsString := 'Visa';
  Open;
 end;
```

1.7.4 Operators

Introduction

Accuracer SQL supports these operator categories:

- Arithmetic operators
- Comparison operators
- Logical operators
- String concatenation operator

Arithmetic Operators

Arithmetic operators perform mathematical operations on two expressions of any of the data types of the numeric data type category.

Operator	Meaning
+ (Add)	Addition
- (Subtract)	Subtraction
* (Multiply)	Multiplication
/ (Divide)	Division

Example:

```
SELECT  (Price * Quantity) AS Total
FROM  Order
```

Bitwise Operators

Bitwise operators perform bitwise operations on the integer data types.

Name	Syntax	Description
SHL	SHL or <<	Bitwise shift left; moves the bits to the left, it discards the far left bit and assigns 0 to the right most bit.
SHR	SHR or >>	Bitwise shift right; moves the bits to the right, discards the far right bit and, if unsigned, assigns 0 to the left most bit, otherwise sign extends.
XOR	XOR or ^	Bitwise exclusive OR; compares two bits and generates a 1 result if the bits are complementary, otherwise it returns 0.
MOD	%, MOD	The same as MOD 41 function.
AND	&	Bitwise AND.
OR	|	Bitwise OR.
NOT	~	Bitwise NOT.

Comparison Operators

Comparison operators test whether or not two expressions are the same. Comparison operators can be used on all expressions except expressions of the **BLOB**, **Memo**, **FmtMemo** or **Graphic** data types.

Operator	Meaning
=, ==	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>, !=	Not equal to

The result of a comparison operator has the Boolean data type, which has three values: TRUE, FALSE, and UNKNOWN. Expressions that return a Boolean data type are known as Boolean expressions.
If an operator that has one or two NULL expressions returns UNKNOWN.
Expressions with Boolean data types are used in the WHERE clause to filter the rows that qualify for the search conditions

Example:

```
SELECT  *
FROM Orders
WHERE (TaxRate > 0)
```

Logical Operators

Logical operators test for the truth of some condition. Logical operators, like comparison operators, return a Boolean data type with a value of TRUE or FALSE.

Operator	Meaning
AND, &&	TRUE if both Boolean expressions are TRUE.
BETWEEN	TRUE if the operand is within a range.
IN	TRUE if the operand is equal to one of a list of expressions.
LIKE	TRUE if the operand matches a pattern.
NOT, !	Reverses the value of any other Boolean operator.
OR	TRUE if either Boolean expression is TRUE.
IS NULL, = NULL	TRUE if Boolean expression is UNKNOWN.
IS NOT NULL, <> NULL	FALSE if Boolean expression is UNKNOWN.
= ""	TRUE if StringArgument IS NULL.
<> ""	FALSE if StringArgument IS NOT NULL.

Examples:

```
SELECT  *
FROM Customer
WHERE (Company LIKE '%Club%')

SELECT  *
FROM Orders
WHERE (ShipToCity IS NOT NULL)

SELECT  *
FROM Orders
WHERE (TaxRate BETWEEN 0 and 5) AND (AmountPaid > 1)

SELECT  *
FROM Orders
WHERE (ShipVIA IN ('UPS', 'DHL'))
```

String Concatenation Operator

The string concatenation operator allows string concatenation with the addition sign (+) or concatenation sign (||), which is also known as the string concatenation operator.

Example:

```
SELECT  (FirstName + ' ' + LastName)  AS  Name
FROM Customers
```

1.7.5 HEX constants

Both C++ and Pascal style of hex constants.

Syntax;
0xff, $ff

Example:

```
drop table test1;
create table test1 (int1 SmallInt, int2 SmallInt);
insert into test1  values (0xff,$ff);
insert into test1  values (0x00,$0f);
insert into test1  values (0x01,$002);
insert into test1  values (0x03,$0002);
select int1, int2, int1 XOR int2 as int3, int2 ^ int1 as int4,
HEX(int1 XOR int2) as str1, HEX(int2 ^ int1,1) as str2, HEX(int2 ^
int1,2) as str3
from test1 order by int1;
```

1.7.6 Functions

There are following types of SQL functions:

1.7.6.1 Aggregate Functions

Name	Syntax	Description
AVG 29	AVG **(** *[DISTINCT] expression* **)**	Returns the average of the values in a group. Null values are ignored.
COUNT 29	COUNT (*[DISTINCT]* expression | *)	Returns the number of items in a group.
GROUP_CONCAT 29	GROUP_CONCAT **(** *[DISTINCT] [ASC] [DESC] expression [, Separator]* **)**	Returns the concatenated string field values for the group of records
MIN 30	MIN (expression)	Returns the minimum value in the expression.
MAX 30	MAX (expression)	Returns the maximum value in the expression.
SUM 30	SUM (*[DISTINCT]* expression)	Returns the sum of all the values in the expression. SUM can be used with numeric columns only. Null values are ignored.

1.7.6.1.1 AVG Function

Returns the average of the values in a group. Null values are ignored.

Syntax;
AVG **(** *[DISTINCT] expression* **)**

Arguments:
expression
Is an expression of the exact numeric or approximate numeric data type category. Aggregate functions and subqueries are not permitted.
If DISTINCT options is specified then SUM calculates only different values of the *expression*.

Examples:
```
SELECT AVG(AmountPaid) FROM Orders WHERE PaymentMethod='Cash'
SELECT AVG(DISTINCT EmpNo) FROM Orders
```

1.7.6.1.2 COUNT Function

Returns the number of items in a group.

Syntax:
COUNT **(** *[DISTINCT] expression* | ***)**

Arguments:
expression
Is an expression of any type except **Blob types**. Aggregate functions and subqueries are not permitted.
If DISTINCT options is specified then COUNT calculates only different values of the *expression*.

*****
Specifies that all rows should be counted to return the total number of rows in a table. COUNT(*****) takes no parameters and cannot be used with DISTINCT. COUNT(*****) does not require an *expression* parameter because, by definition, it does not use information about any particular column. COUNT(*****) returns the number of rows in a specified table without eliminating duplicates. It counts each row separately, including rows that contain null values.

Examples:
```
SELECT COUNT(*) FROM  Orders
SELECT COUNT(OrderN0), ShipVIA FROM Orders GROUP BY ShipVIA
SELECT COUNT(DISTINCT Terms), ShipVIA  FROM Orders GROUP BY ShipVIA
```

1.7.6.1.3 GROUP_CONCAT Function

Returns the concatenated string field values for the group of records.

Syntax:
GROUP_CONCAT **(** *[DISTINCT] [ASC] [DESC] expression [, Separator]* **)**

Arguments:
expression
Is an expression based on the string field.
If DISTINCT options is specified then only different values will be concatenated.

NULL values are always skipped.
ASC or DESC
Specifies if the values should be sorted ascending or descending. ASC is the default setting.
Separator
Specifies the string value that will be used as a separator between values. Comma is the default separator.

You can use GROUP_CONCAT with GROUP BY or without it. In last case all records will be scanned as a single group.

Examples:
```
SELECT GROUP_CONCAT(name) FROM test
SELECT GROUP_CONCAT(DESC name) FROM test
SELECT num, GROUP_CONCAT(DISTINCT DESC name, "; ") FROM test GROUP BY
num
```

1.7.6.1.4 MIN Function

Returns the minimum value in the expression.

Syntax:
MIN **(** *expression* **)**

Arguments:
expression
Is an expression of any type except **Blob types**. Aggregate functions and subqueries are not permitted.

Examples:
```
SELECT MIN(OrderNo) FROM Orders
SELECT MIN(Company) FROM Customer
```

1.7.6.1.5 MAX Function

Returns the maximum value in the expression.

Syntax:
MAX **(** *expression* **)**

Arguments:
expression
Is an expression of any type except **Blob types**. Aggregate functions and subqueries are not permitted.

Example:
```
SELECT MAX(SaleDate) FROM Orders
```

1.7.6.1.6 SUM Function

Returns the sum of all the values in the expression. SUM can be used with numeric columns only. Null values are ignored.

Syntax:

SUM **(** *[DISTINCT] expression* **)**

Arguments:
expression
Is an expression of the exact numeric or approximate numeric data type category. Aggregate functions and subqueries are not permitted.
If DISTINCT options is specified then SUM calculates only different values of the *expression*.

Examples:
```
SELECT SUM(AmountPaid) FROM Orders WHERE PaymentMethod='Visa'
SELECT SUM(DISTINCT EmpNo) FROM Orders
```

1.7.6.2 Date and Time Functions

Name	Syntax	Description
CURRENT_DATE 32	CURRENT_DATE	Returns current system date.
CURRENT_TIME 32	CURRENT_TIME	Returns current system time.
CURRENT_TIMESTAMP 32	CURRENT_TIMESTAMP	Returns current system date and time.
DAY 32	DAY (*expression*)	Returns the day (integer value: 1 - 31) extracted from the date expression.
DAYNAME 33	DAYNAME (*expression*)	Returns the name of the day (string value) extracted from the date expression.
DAYOFWEEK 33	DAYOFWEEK (*expression*)	Returns the day of week (integer value: 1 - Monday, 2 - Tuesday, …, 7 - Sunday) extracted from the date expression.
EXTRACT 33	EXTRACT (*operator* FROM *expression*)	Extracts years, quarters, months, days, hours, minutes, seconds, milliseconds from date, time and datetime expressions.
HOUR 34	HOUR (*expression*)	Returns the hours (integer value: 0 - 23) extracted from the time expression.
MINUTE 34	MINUTE (*expression*)	Returns the minutes (integer value: 0 - 59) extracted from the time expression.
MONTH 34	MONTH (*expression*)	Returns the month (integer value: 1 - 12) extracted from the date expression.
MONTHNAME 35	MONTHNAME (*expression*)	Returns the name of the month (string value) extracted from the date expression.
MSECOND 35	MSECOND (*expression*)	Returns the milliseconds (integer value: 0 - 999) extracted from the time expression.
NOW 32	NOW	Returns current system date and time.
QUARTER 35	QUARTER (*expression*)	Returns the quarter of the year (integer value: 1 - 4) extracted from the date expression.
SECOND 36	SECOND (*expression*)	Returns the seconds (integer value: 0 - 59) extracted from the time expression.
SYSDATE 32	SYSDATE	Returns current system date and time.
TODATE 36	TODATE(*StringValue, DateFormat*)	Converts string to date using specified format.
TOSTRING 37	TOSTRING(*DateValue, DateFormat*)	Converts date to string using specified format.
WEEKDAY 38	WEEKDAY (*expression*)	Returns the day of week (integer value: 1 - Sunday, 2 - Monday, …, 7 - Saturday) extracted from the date expression.
YEAR 39	YEAR (*expression*)	Returns the year (integer value) extracted from the date expression.

1.7.6.2.1 CURRENT_DATE Function

Returns current system date.

Syntax:
CURRENT_DATE

Example:
```
SELECT LastInvoiceDate, CURRENT_DATE as CurDate
FROM Customer
WHERE LastInvoiceDate < NOW
```

1.7.6.2.2 CURRENT_TIME Function

Returns current system time.

Syntax:
CURRENT_TIME

Example:
```
SELECT LastInvoiceDate, CURRENT_TIME as CurTime
FROM Customer
WHERE LastInvoiceDate < NOW
```

1.7.6.2.3 CURRENT_TIMESTAMP, NOW AND SYSDATE Functions

Returns current system date and time.

Syntax:
CURRENT_TIMESTAMP
NOW
SYSDATE

Example:
```
SELECT LastInvoiceDate, NOW as CurDate
FROM Customer
WHERE LastInvoiceDate < NOW
```

1.7.6.2.4 DAY Function

Returns the day (integer value: 1 - 31) extracted from the date expression.

Syntax:
DAY (*expression*)

Arguments:
expression
Is an expression of date, time or datetime types that specifies the source date.

Example:
```
SELECT DAY(LastInvoiceDate) FROM Customer
```

1.7.6.2.5 DAYNAME Function

Returns the name of the day (string value) extracted from the date expression.

Syntax:
DAYNAME (*expression*)

Arguments:
expression
Is an expression of date, time or datetime types that specifies the source date.

Example:
```
SELECT DAYNAME(LastInvoiceDate) FROM Customer
```

1.7.6.2.6 DAYOFWEEK Function

Returns the day of week (integer value: 1 - Monday, 2 - Tuesday, …, 7 - Sunday) extracted from the date expression.

Syntax:
DAYOFWEEK (*expression*)

Arguments:
expression
Is an expression of date, time or datetime types that specifies the source date.

Example:
```
SELECT DAYOFWEEK(LastInvoiceDate) FROM Customer
```

1.7.6.2.7 EXTRACT Function

Extracts years, quarters, months, days, hours, minutes, seconds, milliseconds from date, time and datetime expressions.

Syntax:
EXTRACT (*operator* FROM *expression*)
EXTRACT (*operator*, *expression*)

Arguments:
operator
Specifies the operator for extract operation:

Operator	Description
DAY	Returns the day (integer value: 1 - 31) extracted from the date expression.
DAYNAME	Returns the name of the day (string value) extracted from the date expression.
DAYOFWEEK	Returns the day of week (integer value: 1 - Monday, 2 - Tuesday, …, 7 - Sunday) extracted from the date expression.
HOUR	Returns the hours (integer value: 0 - 23) extracted from the time expression.

MINUTE	Returns the minutes (integer value: 0 - 59) extracted from the time expression.
MONTH	Returns the month (integer value: 1 - 12) extracted from the date expression.
MONTHNAME	Returns the name of the month (string value) extracted from the date expression.
MSECOND	Returns the milliseconds (integer value: 0 - 999) extracted from the time expression.
QUARTER	Returns the quarter of the year (integer value: 1 - 4) extracted from the date expression.
SECOND	Returns the seconds (integer value: 0 - 59) extracted from the time expression.
WEEKDAY	Returns the day of week (integer value: 1 - Sunday, 2 - Monday, …, 7 - Saturday) extracted from the date expression.
YEAR	Returns the year (integer value) extracted from the date expression.

expression
Is an expression of date, time or datetime types that specifies the source date.

Examples:
```
SELECT EXTRACT(YEAR FROM LastInvoiceDate) FROM Customer
SELECT EXTRACT(MONTH, LastInvoiceDate) FROM Customer
```

1.7.6.2.8 HOUR Function

Returns the hours (integer value: 0 - 23) extracted from the time expression.

Syntax:
HOUR (*expression*)

Arguments:
expression
Is an expression of date, time or datetime types that specifies the source date.

Example:
```
SELECT HOUR(LastInvoiceDate) FROM Customer
```

1.7.6.2.9 MINUTE Function

Returns the minutes (integer value: 0 - 59) extracted from the time expression.

Syntax:
MINUTE (*expression*)

Arguments:
expression
Is an expression of date, time or datetime types that specifies the source date.

Example:
```
SELECT MINUTE(LastInvoiceDate) FROM Customer
```

1.7.6.2.10 MONTH Function

Returns the month (integer value: 1 - 12) extracted from the date expression.

Syntax:
MONTH (*expression*)

Arguments:
expression
Is an expression of date, time or datetime types that specifies the source date.

Example:
```
SELECT MONTH(LastInvoiceDate) FROM Customer
```

1.7.6.2.11 MONTHNAME Function

Returns the name of the month (string value) extracted from the date expression.

Syntax:
MONTHNAME (*expression*)

Arguments:
expression
Is an expression of date, time or datetime types that specifies the source date.

Example:
```
SELECT MONTHNAME(LastInvoiceDate) FROM Customer
```

1.7.6.2.12 MSECOND Function

Returns the milliseconds (integer value: 0 - 999) extracted from the time expression.

Syntax:
MSECOND (*expression*)

Arguments:
expression
Is an expression of date, time or datetime types that specifies the source date.

Example:
```
SELECT MSECOND(LastInvoiceDate) FROM Customer
```

1.7.6.2.13 QUARTER Function

Returns the quarter of the year (integer value: 1 - 4) extracted from the date expression.

Syntax:
QUARTER (*expression*)

Arguments:
expression
Is an expression of date, time or datetime types that specifies the source date.

Example:
```
SELECT QUARTER(LastInvoiceDate) FROM Customer
```

1.7.6.2.14 SECOND Function

Returns the seconds (integer value: 0 - 59) extracted from the time expression.

Syntax:
SECOND (*expression*)

Arguments:
expression
Is an expression of date, time or datetime types that specifies the source date.

Example:
```
SELECT SECOND(LastInvoiceDate) FROM Customer
```

1.7.6.2.15 TODATE Function

Converts string to date using specified format.

Syntax:
TODATE(*StringValue, DateFormat*)

Arguments:
StringValue
Is an expression of string or wide string type that specifies the source string.
DateFormat
Is an expression of string or wide string type that specifies date format for the StringValue.

DateFormat strings are composed from specifiers that represent values to be inserted into the formatted string. Some specifiers (such as "d"), simply format numbers or strings. Other specifiers (such as "/") refer to locale-specific strings.

In the following table, specifiers are given in upper case. Case is ignored in formats.

Specifier	Displays
-	Displays date separator '-'.
/	Displays date separator '/'.
.	Displays date separator '.'.
,	Displays date separator ','.
:	Displays date separator ':'.
;	Displays date separator ';'.
'TEXT'	Displays the text that will be included in the result of TOSTRING function without any conversion. The leading and trailing quotes will be omitted.
YYYY or YEAR	Displays the year as a four-digit number (0000-9999)
YY	Displays the year as a two-digit number (00-99)
Q	Displays the quarter of the year (1-4). 1 means months January, February and March, 2 means months April, May and June, 3 means months July, August and September, 4 means months October, November and December.
MONTH	Displays the month as a full name (January-December).
MON	Displays the month as an abbreviation (Jan-Dec).

MM	Displays the month as a number with a leading zero (01-12).
M	Displays the month as a number without a leading zero (1-12).
RM	Displays the month in roman numeric format (I - XII).
DDD	Displays the day of the year (1-366) without a leading zero.
DD	Displays the day of the month (01-31) with a leading zero.
D	Displays the day of the month (1-31) without a leading zero.
DAY	Displays the day as an abbreviation (Sunday-Saturday).
DY	Displays the day as an 3 symbol abbreviation (Sun-Sat).
DW	Displays the day of week (1-7)
HH HH12	Displays the hour with a leading zero (01-12).
HH24	Displays the hour with a leading zero (01-24).
H H12	Displays the hour without a leading zero (1-12).
H24	Displays the hour without a leading zero (1-24).
NN	Displays the minute with a leading zero (00:59).
N	Displays the minute without a leading zero (0:59).
SS	Displays the second with a leading zero (00:59).
S	Displays the second without a leading zero (00:59).
ZZZ	Displays the millisecond with a leading zero (000:999).
Z	Displays the millisecond without a leading zero (0:999).
AMPM	Displays the meridian indicator AM.

Example:

```
SELECT LastInvoiceDate, NOW as CurDate
FROM Customer
WHERE LastInvoiceDate < TODATE('12/16/2002 11:10:30 am','MM/DD/YYYY
hh:nn:ss ampm')
```

1.7.6.2.16 TOSTRING Function

Converts date to string using specified format.

Syntax:
TOSTRING(*DateValue, DateFormat*)

Arguments:
DateValue
Is an expression of date, time or datetime types that specifies the source date.
DateFormat
Is an expression of string or wide string type that specifies date format for conversion DateValue to string.

DateFormat strings are composed from specifiers that represent values to be inserted into the formatted string. Some specifiers (such as "d"), simply format numbers or strings. Other specifiers (such as "/") refer to locale-specific strings.

In the following table, specifiers are given in upper case. Case is ignored in formats.

Specifier	**Displays**
-	Displays date separator '-'.
/	Displays date separator '/'.

.	Displays date separator '.'.
,	Displays date separator ','.
:	Displays date separator ':'.
;	Displays date separator ';'.
'TEXT'	Displays the text that will be included in the result of TOSTRING function without any conversion. The leading and trailing quotes will be omitted.
YYYY or YEAR	Displays the year as a four-digit number (0000-9999)
YY	Displays the year as a two-digit number (00-99)
Q	Displays the quarter of the year (1-4). 1 means months January, February and March, 2 means months April, May and June, 3 means months July, August and September, 4 means months October, November and December.
MONTH	Displays the month as a full name (January-December).
MON	Displays the month as an abbreviation (Jan-Dec).
MM	Displays the month as a number with a leading zero (01-12).
M	Displays the month as a number without a leading zero (1-12).
RM	Displays the month in roman numeric format (I - XII).
DDD	Displays the day of the year (1-366) without a leading zero.
DD	Displays the day of the month (01-31) with a leading zero.
D	Displays the day of the month (1-31) without a leading zero.
DAY	Displays the day as an abbreviation (Sunday-Saturday).
DY	Displays the day as an 3 symbol abbreviation (Sun-Sat).
DW	Displays the day of week (1-7)
HH HH12	Displays the hour with a leading zero (01-12).
HH24	Displays the hour with a leading zero (01-24).
H H12	Displays the hour without a leading zero (1-12).
H24	Displays the hour without a leading zero (1-24).
NN	Displays the minute with a leading zero (00:59).
N	Displays the minute without a leading zero (0:59).
SS	Displays the second with a leading zero (00:59).
S	Displays the second without a leading zero (00:59).
ZZZ	Displays the millisecond with a leading zero (000:999).
Z	Displays the millisecond without a leading zero (0:999).
AMPM	Displays the meridian indicator AM.

Example:

```
SELECT TOSTRING(LastInvoiceDate,"'Today is' mm/dd/yyyy hh24:nn:ss:zzz '
Wow !!!'") Formated_Date, LastInvoiceDate
FROM Customer
```

1.7.6.2.17 WEEKDAY Function

Returns the day of week (integer value: 1 - Sunday, 2 - Monday, …, 7 - Saturday) extracted from the date expression.

Syntax:

WEEKDAY (*expression*)

Arguments:
expression
Is an expression of date, time or datetime types that specifies the source date.

Example:
```
SELECT WEEKDAY(LastInvoiceDate) FROM Customer
```

1.7.6.2.18 YEAR Function

Returns the year (integer value) extracted from the date expression.

Syntax:
YEAR (*expression*)

Arguments:
expression
Is an expression of date, time or datetime types that specifies the source date.

Example:
```
SELECT YEAR(LastInvoiceDate) FROM Customer
```

1.7.6.3 Miscellaneous Functions

Name	Syntax	Description
ISNULL 39	ISNULL (*expression [, replacement]*)	Returns replacement value if expression value is NULL. If replacement is not specified then returns true if the expression is NULL and returns false if expression is not NULL.
LASTAUTOINC 40	LASTAUTOINC(*table_name, column_name*)	Returns the last autoincrement value from a specified table.

1.7.6.3.1 ISNULL Function

Returns replacement value if expression value is NULL. If replacement is not specified then returns true if the expression is NULL and returns false if expression is not NULL.

Syntax:
ISNULL (*expression [, replacement]*)

Arguments:
expression
Is any valid expression.
replacement
Value to replace NULL values. The type of replacement value must be the same as the type of expression value. Use CAST 47 function to covert data types.

Examples:
```
SELECT ISNULL(Addr2,'No Address') FROM Customer
SELECT * FROM Customer WHERE ISNULL(Addr2)
SELECT SUM(ISNULL(AmountPaid,CAST(10.0,CURRENCY))) FROM Orders
```

1.7.6.3.2 LASTAUTOINC Function

The LASTAUTOINC function returns the last autoincrement value from a specified table.

Syntax:
LASTAUTOINC(*table_name, column_name*)

Arguments:
table_name
Is a string constant that specifies table name for getting the last autoincrement value.
column_name
Is a string constant that specifies autoincrement field name for getting the last autoincrement value.

Example:
```
INSERT INTO Employee (Name,DeptID)
VALUES ('John Smith',LASTAUTOINC( Department, ID ))
```

1.7.6.4 Mathematical Functions

Name	Syntax	Description
ABS 41	ABS (x)	Returns the absolute value of x.
SIGN 41	SIGN (x)	SIGN(x) returns the sign of input x as -1, 0, or 1 (negative, zero, or positive respectively).
MOD 41	MOD (x, y)	Returns the integer remainder of x divided by y (same as x%y).
FLOOR 41	FLOOR (x)	Returns the largest integer value that is less than or equal to x.
CEILING 42	CEILING (x) or CEIL (x)	Returns the smallest integer value that is greater than or equal to x.
CUMPROD 42	CUMPROD(x)	Returns the cumulative product of all prior values and current value.
CUMSUM 42	CUMSUM (x)	Returns the cumulative sum of all prior values and current value.
ROUND 43	ROUND (x) or ROUND (x, d)	Returns the value of x rounded to the nearest whole integer. Returns the value of x rounded to the number of decimal places specified by the value d.
TRUNCATE 43	TRUNCATE (x, d) or TRUNC (x, d)	Returns the number X, truncated to D decimals. If D is 0, the result will have no decimal point or fractional part.
POWER 43	POWER (x, y) or POW (x, y)	Returns the value of x raised to the power of y.
HEX 44	HEX(X [, MODE])	If X is a number, returns a string representation of the hexadecimal value of X, where X is integer. If X is a string, returns a hexadecimal string of X where each character in X is converted to 2 hexadecimal digits. MODE: 0 - just convert to hex (default), 1 - Pascal hex style ($FF), 2 - C++ hex style (0xff).
RAND, RANDOM 44	RAND ( ) or RAND (n)	Returns a random floating-point value in the range 0 to 1.0. If an integer argument N is specified, it is used as maximum value and result will be integer 0 <= X < N

1.7.6.4.1 ABS Function

Returns the absolute value of x.

Syntax;
ABS (x)

Arguments:
x
Is an expression of the numeric data type.

Examples:
```
SELECT abs(int1) as int_val, abs(float1) as float_Val from test1
```

1.7.6.4.2 SIGN Function

Returns the concatenated string field values for the group of records
Returns the sign of input x as -1, 0, or 1 (negative, zero, or positive respectively).

Syntax;
SIGN (x)

Arguments:
x
Is an expression of the numeric data type.

Examples:
```
SELECT sign(int1) as int_val, sign(float1) as float_Val from test1
```

1.7.6.4.3 MOD Function

Returns the integer remainder of x divided by y (the same as x%y).

Syntax;
MOD (x)

Arguments:
x
Is an expression of the numeric data type.

Examples:
```
SELECT int1 % int2 as int_Val2, int1 MOD int2 as int_Val3 from test1
```

1.7.6.4.4 FLOOR Function

Returns the largest integer value that is less than or equal to x.

Syntax;
FLOOR (x)

Arguments:
x

Is an expression of the numeric data type.

Examples:
```
SELECT FLOOR(int1) as int_val, FLOOR(float1) as float_Val from test1
```

1.7.6.4.5 CEILING Function

Returns the absolute value of x.

Syntax;
CEILING (x)

Arguments:
x
Is an expression of the numeric data type.

Examples:
```
SELECT CEIL(int1) as int_val, CEILING(float1) as float_Val from test1
```

1.7.6.4.6 CUMSUM Function

Returns the cumulative sum of all prior values and current value.

Syntax;
CUMSUM (x)

Arguments:
x
Is an expression based on numeric field values.

Examples:
```
SELECT Cost,CUMSUM(Cost) FROM Orders
```

1.7.6.4.7 CUMPROD Function

Returns the cumulative product of all prior values and current value.

Syntax;
CUMPROD(x)

Arguments:
x
Is an expression based on numeric field values.

Examples:
```
SELECT Cost,CUMPROD(Cost) FROM Orders
```

1.7.6.4.8 ROUND Function

Returns the value of *x* rounded to the nearest whole integer or to the number of decimal places specified by the value *d*.

Syntax;
ROUND (x) or
ROUND (x, d)

Arguments:
x
Is an expression of the numeric data type.
d
Is an expression of the integer data type.

Examples:
```
SELECT float1, ROUND(float1) as float_Val1, ROUND(float1,1) as
float_Val2, ROUND(float1,2) as float_Val3 from test1
```

1.7.6.4.9 TRUNCATE Function

Returns the number X, truncated to D decimals. If D is 0, the result will have no decimal point or fractional part.

Syntax;
TRUNCATE (x)

Arguments:
x
Is an expression of the numeric data type.
d
Is an expression of the integer data type.

Examples:
```
SELECT float1, TRUNCATE(float1) as float_val1, TRUNC(float1,1) as
float_val2, TRUNC(float1,2) as float_val3 from test1
```

1.7.6.4.10 POWER Function

Returns the value of *x* raised to the power of *y*.

Syntax;
POWER (x, y) or
POW (x, y)

Arguments:
x, y
Are an expressions of the numeric data type.

Examples:
```
SELECT POWER(int1,int2) as exp1, POW(float1,int2) as exp2 from test1
```

1.7.6.4.11 HEX Function

If X is a number, returns a string representation of the hexadecimal value of *X*, where *X* is integer. If *X* is a string, returns a hexadecimal string of *X* where each character in *X* is converted to 2 hexadecimal digits.

Syntax;
HEX (*X* [, *MODE*])

Arguments:
x
Is an expression of the numeric data type.

MODE
0 - just convert to hex (default),
1 - Pascal hex style ($FF),
2 - C++ hex style (0xff).

Examples:
```
select HEX(int1 XOR int2) as str1, HEX(int2 ^ int1,1) as str2, HEX(int2
^ int1,2) from test1
```

1.7.6.4.12 RANDOM Function

Returns a random floating-point value in the range 0 to 1.0. If an integer argument *n* is specified, it is used as maximum value and result will be integer *X*, where 0 <= *X* < *n.*

Syntax;
RAND ( ) or
RAND (*n*) or
RANDOM ( ) or
RANDOM (*n*)

Arguments:
n
Is an expression of the integer data type.

Examples:
```
select test1.*, RAND(100000) as rnd from memory test1 order by num
desc,id
select test1.*, RANDOM as rnd from memory test1 order by num desc,id
```

1.7.6.5 String Functions

Name	Syntax	Description
LENGTH 45	LENGTH (*expression*)	Returns the number of characters in a string excluding the null terminator.
LOWER 45	LOWER (*expression*)	Returns a character expression after converting uppercase character data to lowercase.

LTRIM 45	LTRIM (*expression*)	Returns a character expression after removing leading blanks.
POS 46	POS (*substring*, expression)	Returns the index value of the first character in a specified substring that occurs in a given string. Pos is case-sensitive.
RTRIM 46	RTRIM (*expression*)	Returns a character string after truncating all trailing blanks.
SUBSTRING 46	SUBSTRING (expression, *startindex [, length]*)	Returns a substring of a string.
TRIM 47	TRIM (*expression*)	Returns a character string after truncating all leading and trailing blanks.
UPPER 47	UPPER (*expression*)	Returns a character expression with lowercase character data converted to uppercase.

1.7.6.5.1 LENGTH Function

Returns number of characters in a string excluding the null terminator.

Syntax:
LENGTH (*expression*)

Arguments:
expression
Is a an expression of string or wide string type.

Example:
```
SELECT * FROM Customer
WHERE LENGTH(Company) > 5
```

1.7.6.5.2 LOWER Function

Returns a character expression after converting uppercase character data to lowercase.

Syntax:
LOWER (*expression*)

Arguments:
expression
Is an expression of string or wide string types.

Example:
```
SELECT LOWER(Company) FROM Customer
```

1.7.6.5.3 LTRIM Function

Returns a character expression after removing leading blanks.

Syntax:
LTRIM (*expression*)

Arguments:
expression
Is an expression of string or wide string types.

Example:
```
SELECT LTRIM(Company) FROM Customer
```

1.7.6.5.4 POS Function

Returns the index value of the first character in a specified substring that occurs in a given string. Pos is case-sensitive.

Syntax:
POS (*substring*, *expression*)

Arguments:
substring
Is a an expression of string or wide string type that specifies substring for searching in the specified string.
expression
Is a an expression of string or wide string type that specifies source string.

Example:
```
SELECT * FROM Customer
WHERE Pos('Blue',Company) > 0
```

1.7.6.5.5 RTRIM Function

Returns a character string after truncating all trailing blanks.

Syntax:
RTRIM (*expression*)

Arguments:
expression
Is an expression of string or wide string types.

Example:
```
SELECT RTRIM(Company) FROM Customer
```

1.7.6.5.6 SUBSTRING Function

Returns a substring of a string.

Syntax:
SUBSTRING (*expression, startindex [, length]*)

Arguments:
expression
Is a an expression of string or wide string type.
startindex
Is a constant that specifies the character position at which the extracted substring starts within the original string.
length
Is a constant that specifies number of characters being extracted from source string.

Example:
```
SELECT SUBSTRING(Company,2,5)
```

```
FROM Customer
```

1.7.6.5.7 TRIM Function

Returns a character string after truncating all leading and trailing blanks.

Syntax:
TRIM (*expression*)

Arguments:
expression
Is an expression of string or wide string types.

Example:
```
SELECT TRIM(Company) FROM Customer
```

1.7.6.5.8 UPPER Function

Returns a character expression with lowercase character data converted to uppercase.

Syntax:
UPPER (*expression*)

Arguments:
expression
Is an expression of string or wide string types.

Example:
```
SELECT UPPER(Company) FROM Customer
```

1.7.6.6 Type Conversion Functions

Name	Syntax	Description
CAST 47	CAST(*value, data_type*)	The CAST function converts a specified value to the specified data type.
TOBLOB 48	TOBLOB(*value [, format]*)	The TOBLOB function converts a specified string value to the BLOB value.

1.7.6.6.1 CAST Function

The CAST function converts a specified value to the specified data type.

Syntax:
CAST(*value, data_type*)

Arguments:
value
Is an expression of any valid data type.
data_type
Is a constant that specifies data type for converting the value specified by Value.
CAST function can be used with the following data types:

Data type	Description
AutoInc	Auto incremental 32-bit unsigned integer.
BCD	Floating point number.
Currency	Floating point number.
Date	Date value.
DateTime	DateTime value.
Float	Floating point number.
Integer	32-bit signed integer.
LargeInt	64-bit signed integer.
Logical	Boolean value.
SmallInt	16-bit signed integer
String	Fixed length string (may be up to 2^32 symbols)
Time	Time value.
WideString	Fixed length Unicode string (may be up to 2^32 symbols)
Word	16-bit unsigned integer.

1.7.6.6.2 TOBLOB Function

The TOBLOB function converts a specified string value to the BLOB value.

Syntax:
TOBLOB(*value [, format]*)

Arguments:
value
Is a string value that can be converted to a BLOB value using specified format.
format
Two formats are supported:
MIME64 - MIME64 standard format (used in e-mail)
HEX - upper case hexadecimal numbers
Default format is MIME64 (typically provides smaller string length).

Example:
```
INSERT INTO jpeg VALUES (
      'AidAim',
      TOBLOB ('QWlkQWltIFNvZnR3YXJlDQpIZXJlIHRvIEhlbHANCg==',MIME64),
      NULL, 1);
```

1.7.7 SELECT Statement

Introduction

The SELECT statement is used to retrieve data from tables.

Syntax

SELECT [DISTINCT | ALL] [TOP n [, first_row_number]]
* | column [AS correlation_name | correlation_name], [column...]
[INTO destination_table]
FROM table_reference [AS correlation_name | correlation_name] [PASSWORD 'password_string']
[[[[NATURAL] [INNER | [LEFT | RIGHT | FULL] OUTER JOIN] table_reference [AS

correlation_name | correlation_name]
[ON join_condition] | USING (join columns)]
[WHERE predicates]
[GROUP BY group_fields_list]
[HAVING predicates]
[ORDER BY order_list]
[UNION [ALL] [CORRESPONDING [BY (column_list)]] SELECT...]
[EXCEPT | MINUS [ALL] [CORRESPONDING [BY (column_list)]] SELECT...]
[INTERSECT [ALL] [CORRESPONDING [BY (column_list)]] SELECT...]

The SELECT clause specifies list of retrieved columns. Use asterisk to select all columns.
The ALL option (Default) retrieves all rows from specified tables.
The DISTINCT option retrieves only different rows. Here is a simple example that retrieves all different contacts from table Customer.

SELECT DISTINCT Contact FROM Customer

The TOP option specifies that only the first *n* rows are to be output from the query result set. If first_row_number specified then n rows starting from this row number will be retreived.
The INTO clause specifies table name for storing data retrieved by SELECT statement.
The FROM clause specifies columns to be retrieved from table(s).
The WHERE clause specifies filtering conditions for the SELECT statement.
The GROUP BY clause divides a result set into groups.

SELECT CustNo, SUM(ItemsTotal)
FROM Orders
GROUP BY CustNo

The HAVING clause specifies a search condition for a group or an aggregate.

ORDER BY clause

Syntax:
1) ORDER BY <order_list>
or
2) ORDER BY INDEX IndexName

<order_list> ::= [TableName.]FieldName [ASC | DESC] [NOCASE]

The ORDER BY clause specifies the sorting order for rows retrieved by the query.

Syntax 1 allows you to specify fields for sorting.
If options are not specified the sorting order will be ascending and case-sensitive.
Option ASC means ascending sorting for this field.
Option DESC means descending sorting for this field.
Option NOCASE means case-insensitive sorting for string field.

The following example shows how to save rows from table Customers to a new table:

SELECT CustNo, Contact, Company, City, Country
INTO NewCustomer
FROM Customer
ORDER BY Country DESC, City DESC NOCASE, Company NOCASE, Contact

Syntax 2 allows you to specify the name of existing index for sorting.
This syntax can be used only for queries on single table.

Here is an example:

SELECT * from Customer_findKey ORDER BY INDEX ByCompany

Join clauses, Natural and Using operators

There are three types of joins that can be used in FROM clause to perform relational joins: Cartesian, Inner and Outer.

A Cartesian join combines source tables without any correlation. The result of such join is a table containing all columns from source tables and all combinations of all rows from source tables. For example, if first table contains 5 records and second table contains 10 records the result of Cartesian join of these tables will contain 50 records. The syntax is as follows:

FROM table_reference, table_reference [,table_reference...]

Here is an example:

SELECT * FROM Members,Departments

An Inner join includes all combined rows from source tables that have common values of specified columns.
An Outer join includes all combined rows from source tables that have common values of specified columns and rows from left, right or both source tables that does not have corresponding rows in other source table. Thus Outer joins can be LEFT, RIGHT or FULL.
The non-corresponding rows from left table contain NULL values for columns from right table and vice versa.
One of the main advantages of Accuracer is that it supports ALL types of JOINS: Cartesian, Inner, Left Outer, Right Outer, Full Outer.
Moreover, the Accuracer provides high performance on joins due to its flexible and well-designed architecture that excludes all unnecessary data transfers.

There are three ways of specifying inner or outer joins:

1) The common columns are specified in ON clause:
FROM table_reference [INNER | LEFT | RIGHT | FULL] JOIN table_reference
ON predicate
[[INNER | LEFT | RIGHT | FULL] JOIN table_reference ON predicate...]

2) The common columns are specified by Using operator (all source tables should contain these columns):
FROM table_reference [INNER | LEFT | RIGHT | FULL] JOIN table_reference USING (column_name[,column_name...])
[[INNER | LEFT | RIGHT | FULL] JOIN table_reference USING (column_name[,column_name...])...]

3) The common columns are all columns from source tables that have the same names.
FROM table_reference NATURAL [INNER | LEFT | RIGHT | FULL] JOIN table_reference
[NATURAL [INNER | LEFT | RIGHT | FULL] JOIN table_reference...]

Here are some examples of joins:

SELECT Contact, Customer.CustNo, Company, Orders.OrderNo, Orders.CustNo

```
FROM Customer INNER JOIN Orders
ON (Customer.CustNo = Orders.CustNo)
WHERE Contact LIKE 'E%'
ORDER BY Contact,Orders.CustNo,Orders.OrderNo

SELECT Contact, Customer.CustNo, Company, Orders.OrderNo, Orders.CustNo
FROM Customer INNER JOIN Orders Using (CustNo)
ORDER BY Contact,Orders.CustNo,Orders.OrderNo

SELECT cb.*
FROM Customer_Base cb NATURAL INNER JOIN Customer_Base

SELECT Contact, Customer.CustNo, Company, Orders.OrderNo, Orders.CustNo
FROM Customer NATURAL LEFT JOIN Orders
WHERE State IS NOT NULL
ORDER BY Contact,Orders.CustNo,Orders.OrderNo

SELECT Contact, Customer.CustNo, Company, Orders.OrderNo, Orders.CustNo
FROM Customer NATURAL RIGHT JOIN Orders
WHERE State IS NOT NULL
ORDER BY Contact,Orders.CustNo,Orders.OrderNo

SELECT Contact, Customer.CustNo, Company, Orders.OrderNo, Orders.CustNo
FROM Customer NATURAL FULL JOIN Orders
WHERE State IS NOT NULL
ORDER BY Contact,Orders.CustNo,Orders.OrderNo
```

UNION clause

Combines the results of two or more queries into a single result set consisting of all the rows belonging to all queries in the union. This is different from using joins that combine columns from two tables. The syntax is:

[UNION [ALL] [CORRESPONDING [BY (column_list)]] SELECT...]

Three basic rules for combining the result sets of two queries with UNION are:

- If CORRESPONDING option is not specified then the number and the order of the columns must be identical in both combined queries.
- If CORRESPONDING option is specified then the listed columns must exist in both queries
- The data types must be identical.

ALL option incorporates all rows into the results, including duplicates. If not specified, duplicate rows are removed.

Example:

```
SELECT Company FROM customer_Base
UNION
SELECT Company FROM customer_Filter
UNION
SELECT Company FROM customer_Range
```

EXCEPT (MINUS) clause

Returns the result set consisting of the rows belonging to the first query, excluding the rows having

identical ones in the second query, optionally retaining duplicates.

[EXCEPT [ALL] [CORRESPONDING [BY (column_list)]] SELECT...]

Three basic rules for EXCEPT are:

- If CORRESPONDING option is not specified then the number and the order of the columns must be identical in both queries.
- If CORRESPONDING option is specified then the listed columns must exist in both queries
- The data types must be identical.

ALL option incorporates all rows into the results, including duplicates. If not specified, duplicate rows are removed.

Example:

```
SELECT * FROM customer_Range
EXCEPT CORRESPONDING BY (Company)
SELECT * FROM customer_Filter
```

INTERSECT clause

Returns the result set consisting of the rows belonging to the first query having identical ones in the second query, optionally retaining duplicates.

[INTERSECT [ALL] [CORRESPONDING [BY (column_list)]] SELECT...]

Three basic rules for INTERSECT are:

- If CORRESPONDING option is not specified then the number and the order of the columns must be identical in both queries.
- If CORRESPONDING option is specified then the listed columns must exist in both queries
- The data types must be identical.

ALL option incorporates all rows into the results, including duplicates. If not specified, duplicate rows are removed.

Example:

```
SELECT * FROM customer_Range
INTERSECT CORRESPONDING BY (Company)
SELECT * FROM customer_Filter
```

WHERE Clause

Specifies the conditions that must be satisfied for all records retrieved by the query.
WHERE clause can include any supported functions and operators excepting aggregative functions.
Accuracer supports only **uncorrelated subqueries**, i.e. you cannot use fields of the parent query in a subquery.

Example of correlated query - will not work in Accuracer:

```
SELECT field1 FROM table1 T1
WHERE T1.field2 = (SELECT MAX(field1) FROM table2 T2 WHERE T2.field2 = T1.field3);
```

Look at examples in Utils\Bin\SQLConsole\SQL\SubQuery folder.

Example:

```
SELECT * from Jpeg
WHERE ID = (SELECT MIN(ID) from jpeg)

SELECT * FROM orders
WHERE CustNo IN
 (SELECT DISTINCT CustNo FROM customer WHERE (Company LIKE 'S%') and (CustNo <
2500))
 ORDER BY CustNo

SELECT Count(*) as ROW_COUNT FROM jpeg
WHERE EXISTS
 (SELECT * FROM jpeg WHERE (Name LIKE 'A%'))
```

1.7.8 INSERT Statement

Introduction

The INSERT statement is used to add one or more rows of data in a table.

Syntax

```
INSERT INTO table_reference [password 'pass'] [(columns_list)]
VALUES (update_values)
```

Use the INSERT statement to add rows of data to a single table.
MEMORY option specifies that in-memory table will be created.
The INTO clause specifies the table to receive the inserted data. The columns list is a comma-separated list, enclosed in parentheses, of columns in the table and is optional. If columns list is not specified the data will be inserted into all columns of the table.
The VALUES clause specifies data to be inserted to the table.

Here is an example how to add records using INSERT statement:

```
INSERT INTO Customer (CustNo,Company, City, State, Contact, LastInvoiceDate)
VALUES (5555,'AidAim Software','Phoenix','AZ','Ella Perelman','10/15/2002')
```

To add rows to one table that are retrieved from another table, omit the VALUES keyword and use a subquery as the source for the new rows:

```
INSERT INTO Customer_Sort
SELECT * FROM Customer_Share
```

1.7.9 UPDATE Statement

Introduction

The UPDATE statement is used to modify one or more existing rows in a table.

Syntax

UPDATE table_reference [Password "password_value"]
SET column_ref = update_value [,column_ref = update_value...]
[WHERE condition]

Use the UPDATE statement to modify one or more column values in one or more existing rows in a single table.
Use a table reference in the UPDATE clause to specify the table to receive the modified data.
MEMORY option specifies that in-memory table will be created.

The SET clause is a comma-separated list of update expressions for the UPDATE statement. The syntax is as follows:

SET column_ref = update_value [,column_ref = update_value...]

Use SET clause to specify columns to update data and a new values for them.

The WHERE clause specifies filtering conditions for the UPDATE statement. The syntax is as follows:

WHERE condition

Use a WHERE clause to update only records that meets specified conditions.

Here is an example:

UPDATE Members SET FirstName = 'New Name' WHERE ID >= 3

1.7.10 DELETE Statement

Introduction

The DELETE statement is used to delete one or more rows of data from a table.

Syntax

DELETE
FROM table_reference [PASSWORD 'password_string']
[WHERE predicates]

MEMORY option specifies that in-memory table will be created.
The FROM clause specifies the table to use for the DELETE statement. The syntax is as follows:
FROM table_reference

The WHERE clause specifies filtering conditions for the UPDATE statement. The syntax is as follows:

WHERE predicates

Use a WHERE clause to update only records that meets specified conditions.

Here is an example

DELETE FROM Members WHERE ID >= 3

1.7.11 CREATE DATABASE Statement

Introduction

The CREATE DATABASE statement is used to create a new database.

Syntax

```
CREATE DATABASE
        FILE "file_name" [ PAGESIZE {128..65535} ]  [ MAXSESSIONSCOUNT {1..2147483648} ]
       |
        MEMORY database_name
```

Use CREATE DATABASE to create database with specified parameters.

You must specify FILE or MEMORY option to create disk or in-memory database, otherwise exception will be raised.

For disk database, file_name parameter is required. File name must be quoted by single quote (') or double quote (").
If PAGESIZE or MAXSESSIONCOUNT is missed, default value will be used.

In case of in-memory database, you must specify the name of database only which can be quoted, double-quoted, or non-quoted.

Example 1.
CREATE DATABASE FILE "c:\temp\test.adb" PAGESIZE 8192 MAXSESSIONSCOUNT 10
- creates disk database with the parameters specified.

Example 2.
CREATE DATABASE MEMORY MemDB1
- creates in-memory database with the name MemDB1.

1.7.12 DROP DATABASE Statement

Introduction

The DROP DATABASE statement is used to delete the database.

Syntax

```
DROP DATABASE
        FILE "file_name"   |   MEMORY database_name
```

Use DROP DATABASE to delete database and all its data.

You must specify FILE or MEMORY option to delete disk or in-memory database, otherwise exception will be raised.

For disk database, file_name parameter is required. File name must be quoted by single quote (') or double quote (").

In case of in-memory database, you must specify the name of the existing database which can be quoted, double-quoted, or non-quoted.

Example 1.
DROP DATABASE FILE "c:\temp\test.adb"
- deletes database and its file c:\temp\test.adb.

Example 2.
DROP DATABASE MEMORY MemDB1
- deletes in-memory database with the name MemDB1.

1.7.13 CREATE TABLE Statement

Introduction

The CREATE DATABASE statement is used to create a new DATABASE.

Syntax

```
CREATE DATABASE DATABASE_name (
  column_name data_type [(dimensions)] |
                    AutoInc[([data_type ]
                              [ , INCREMENT integer ]
                              [ , INITIALVALUE integer ]
                              [ , MAXVALUE integer | NOMAXVALUE ]
                              [ , MINVALUE integer   | NOMINVALUE  ]
                              [ , CYCLED | NOCYCLED ]
                             )]

    [ BLOBBLOCKSIZE {1..4294967295} ]
    [ BLOBCOMPRESSIONALGORITHM {NONE | ZLIB | BZIP | PPM} ]
    [ BLOBCOMPRESSIONMODE     {0 .. 9} ]

    [ DEFAULT {const | NULL} ]
    [ NOT NULL | NULL ]
    [ PRIMARY [KEY] | UNIQUE  [ ASC | DESC ]  [ CASE | NOCASE ]]
    [MINVALUE value | NOMINVALUE ]
    [MAXVALUE value | NOMAXVALUE ]
  [,column_name …]

  [ , PRIMARY KEY [key name] (column_name  [ ASC | DESC ]  [ CASE | NOCASE ]
                                        [ {, column_name  [ ASC | DESC ]  [ CASE | NOCASE ] } ...] ) ]
  [ , FOREIGN KEY [key name] (column_name [ { ,column_name } ... ])
      REFERENCES DATABASE_name [MATCH FULL | MATCH PARTIAL]
      [ON DELETE <CASCADE | SET NULL |  SET DEFAULT | NO ACTION>]
      [ON UPDATE <CASCADE | SET NULL |  SET DEFAULT | NO ACTION>] ]
)
```

DATABASE_name = [MEMORY] name_of_the_DATABASE.

Use CREATE DATABASE to create DATABASE with specified structure.
The DATABASE_name is the name of the DATABASE to be created. If MEMORY option is specified then an in-memory DATABASEwill be created.
The column_name is a name of the column. The data_type can be one of the following:

Value	Description	Corresponding TFieldType
Char, FixedChar	Fixed character field	ftFixedChar
Varchar, Varchar2,	Character or variable length	ftString

String	string field	
WideChar, FixedWideChar	Fixed wide character field	ftWideString
WideVarchar, WideString	Wide character or variable length wide string field	ftWideString
Shortint, SignedInt8	8-bit integer field	ftSmallint
Smallint, SignedInt16	16-bit integer field	ftSmallint
Integer, SignedInt32	32-bit integer field	ftInteger
Largeint, Int64, SignedInt64	64-bit integer field	ftLargeint
Byte, UnsignedInt8	Byte field	ftWord
Word, UnsignedInt16	16-bit unsigned integer field	ftWord
Cardinal, UnsignedInt32	32-bit unsigned integer field	ftLargeint
AutoInc, AutoincInteger	Auto-incrementing 32-bit integer counter field	ftAutoinc
AutoIncShortint	Auto-incrementing 8-bit integer counter field	ftAutoinc
AutoIncSmallint	Auto-incrementing 16-bit integer counter field	ftAutoinc
AutoIncLargeint	Auto-incrementing 64-bit integer counter field	ftAutoinc
AutoIncByte	Auto-incrementing byte counter field	ftAutoinc
AutoIncWord	Auto-incrementing 16-bit unsigned integer counter field	ftAutoinc
AutoIncCardinal	Auto-incrementing 32-bit unsigned integer counter field	ftAutoinc
Single	Single floating-point numeric field	ftFloat
Float, Double	Double floating-point numeric field	ftFloat
Extended	Extended floating-point numeric field	ftFloat
Boolean, Logical, Bool, Bit	Boolean field	ftBoolean
Currency, Money	Money field	ftCurrency
Date	Date field	ftDate
Time	Time field	ftTime
DateTime	Date and time field	ftDateTime
TimeStamp	Date and time field accessed through dbExpress	ftTimeStamp
Bytes	Fixed number of bytes (binary storage)	ftBytes
VarBytes	Variable number of bytes (binary storage)	ftVarBytes
Blob	Binary Large OBject field	ftBlob
Graphic	Bitmap field	ftGraphic
Memo, Clob	Text memo field	ftMemo
FormattedMemo, FmtMemo	Formatted text memo field	ftFmtMemo
WideMemo, WideClob	Unicode text memo field	ftMemo

The dimensions is a size of the column value in bytes. Use it with bytes, string or wide string data

types.
Use NOT NULL option to specify columns with required not empty values.

Specify compression level for storing BLOB field values (BLOB,FmtMemo,Memo,Graphic) via BlobCompressionAlgorithm and BlobCompressionMode options.
The BlobBlockSize is the size in bytes of BLOB data block which is used by database engine in read / write operations with BLOB fields. Minimum value 1 byte, default value 100 Kb.

Here is an example:

```
CREATE DATABASE Test
(
 ID AutoInc PRIMARY KEY,
 Text String(500),
 Numeric Float,
 Money Currency,
 CurrentDate Date,
 Picture Graphic BlobCompressionAlgorithm ZLIB BlobCompressionMode 1
);
```

Note:
If the DATABASE with the specified name already exists, CREATE DATABASE will raise an exception.
DATABASE, column and index names can be specified in square brackets ([]). Thus you can use reserved words (like DATABASE) and special symbols (like ' ') in DATABASE, column and index names.

1.7.14 ALTER TABLE Statement

Introduction

The ALTER TABLE statement is used to modify a structure of the existing table.

Syntax

```
ALTER TABLE  table_name ADD [COLUMN]
(
  column_name data_type [(dimensions)] [NOT NULL]
  [,column_name data_type [(dimensions)] [NOT NULL]...]
  [,PRIMARY KEY (column_name [, column_name...])]
)
ALTER TABLE table_name ADD
(
  [ PRIMARY KEY [key name] (column_name  [ ASC | DESC ]  [ CASE | NOCASE ]
                                        [, column_name...]) ]
  [ FOREIGN KEY [key name] (column_name [ { ,column_name } ... ])
      REFERENCES table_name [MATCH FULL | MATCH PARTIAL]
      [ON DELETE <CASCADE | SET NULL |  SET DEFAULT | NO ACTION>]
      [ON UPDATE <CASCADE | SET NULL |  SET DEFAULT | NO ACTION>] ]
)

ALTER TABLE TableName  <MODIFY> | <ALTER [COLUMN]> (
  column_name data_type [(dimensions)] |
                    AutoInc[([data_type ]
```

```
                                [ , INCREMENT integer ]
                                [ , INITIALVALUE integer ]
                                [ , MAXVALUE integer | NOMAXVALUE ]
                                [ , MINVALUE integer   | NOMINVALUE  ]
                                [ , CYCLED | NOCYCLED ]
                               )]

    [ BLOBBLOCKSIZE {1..4294967295} ]
    [ BLOBCOMPRESSIONALGORITHM {NONE | ZLIB | BZIP | PPM} ]
    [ BLOBCOMPRESSIONMODE     {0 .. 9} ]

    [ DEFAULT {const | NULL} | DROP DEFAULT ]
    [ NOT NULL | NULL ]
    [ PRIMARY [KEY] | UNIQUE  [ ASC | DESC ]  [ CASE | NOCASE ]]
    [MINVALUE value | NOMINVALUE ]
    [MAXVALUE value | NOMAXVALUE ]
)


ALTER TABLE table_name DROP [COLUMN]
(
column_name [,column_name...]
)

ALTER TABLE table_name DROP CONSTRAINT constraint_name [CASCADE | RESTRICT]

ALTER TABLE TableName RENAME [COLUMN] OldName [ TO ] NewName

ALTER TABLE TableName  RENAME TO NewName
or
RENAME TABLE TableName TO NewName
```

Use ALTER TABLE to modify a structure of the existing table.
The ADD clause is used to add new columns to the table.
The ALTER or MODIFY clauses are used to modify columns definitions.

Note: Accuracer always tries to keep existing values for the modified columns when it possible. However, some type of conversions causes data losses - for example, if you will convert string column to the integer one, all values that cannot be converted to integer will be replaced with NULL values.

The DROP clause is used to remove columns from the table.

Here are some examples:

```
ALTER TABLE Test DROP (Numeric);

ALTER TABLE Test ADD (NewField WideString(500));

ALTER TABLE Test DROP CONSTRAINT PK CASCADE

ALTER TABLE Emp ADD FOREIGN KEY FKDeptID (DeptID) REFERENCES Dept MATCH FULL
 ON DELETE CASCADE ON UPDATE SET DEFAULT
```

1.7.15 DROP TABLE Statement

Introduction

The DROP TABLE statement is used to delete table from the database.

Syntax

DROP TABLE table_name [CASCADE | RESTRICT]

CASCADE option forces to drop all other objects in database referencing this table (like foreign key constraints).

Here is an example:

DROP TABLE Test

Note:
If the table does not exist, DROP TABLE will not raise an exception.

1.7.16 CREATE INDEX Statement

Introduction

The CREATE INDEX statement is used to create new index in a table.

Syntax

```
CREATE [UNIQUE] INDEX [IF NOT EXISTS] index_name ON table_name
(
  field_name [ASC | DESC] [CASE | NOCASE]
  [,field_name...]
)
```

Use CREATE INDEX to create index in a table. Indexes are used to increase search and sorting speed. Indexes decrease performance of inserting, updating and deleting data.

The UNIQUE option specifies restriction on inserting rows to a table with duplicate columns values. It means that all rows in a table have unique combination of index columns values.

The IF NOT EXISTS option specifies that index should be created only if it does not exists in this table.

The ASC option specifies ascending order, the DESC option specifies descending order. The default value is ASC.
The CASE option specifies case-sensitive index, the NOCASE specifies case-insensitive index. The default value is CASE.

Examples:

```
CREATE UNIQUE INDEX Text_Index ON Test
(
 Text DESC NOCASE,
 ID
)
```

```
CREATE UNIQUE INDEX Text_Index ON MEMORY [Test MEMORY TABLE]
(
 [My Text] DESC NOCASE,
 ID
)
```

Note:
Table, column and index names can be specified in square brackets ([]). Thus you can use reserved words (like TABLE) and special symbols (like ' ') in table, column and index names.

1.7.17 DROP INDEX Statement

Introduction

The DROP INDEX statement is used to delete index from the table.

Syntax

DROP INDEX [IF EXISTS] table_name.index_name

The IF EXISTS option specifies that index should be dropped only if it exists in this table.

Here is an example:

DROP INDEX Test.Text_Index

1.7.18 START TRANSACTION Statement

Introduction

The START TRANSACTION statement is used to start transaction in the database.

For more information read Transactions topic in Developer's Guide.

Syntax

START TRANSACTION

Here is an example:

```
START TRANSACTION;
INSERT INTO Table1 (NAME) VALUES('aaa');
UPDATE Table2 SET Field1 = Field1 + 1;
INSERT INTO Table1 (NAME) VALUES('bb');
COMMIT;
```

1.7.19 COMMIT Statement

Introduction

The COMMIT statement is used to finish current transaction and write all changes to the database file.
NOFLUSH option specifies that file buffers should not be flushed after commit finished.
If this option is specified the COMMIT will work a little faster and all changes will be saved to the database file later, by separate OS process. If this option is not specified the COMMIT will save all changes to the database file and perform flusing of the file buffers, so after executing all data will be save to the database file.

For more information read Transactions topic in Developer's Guide.

Syntax

COMMIT [NOFLUSH]

Here is an example:

```
START TRANSACTION;
INSERT INTO Table1 (NAME) VALUES('aaa');
UPDATE Table2 SET Field1 = Field1 + 1;
INSERT INTO Table1 (NAME) VALUES('bb');
COMMIT NOFLUSH;
```

1.7.20 ROLLBACK Statement

Introduction

The ROLLBACK statement is used to abort current transaction and discard all changes made by this transaction.

For more information read Transactions topic in Developer's Guide.

Syntax

ROLLBACK

Here is an example:

```
ROLLBACK;
```

1.8 Multi-User and Multi-Thread, Locking Mechanism and Transactions

1.8.1 Multi-User and Multi-Thread Support

Introduction

Accuracer can be used both in multi-user and / or multi-therad environments as well as in single user mode.
The only restriction is a **SU** edition that can be used only in multi-thread environment, but not in

multi-user.
The **Trial** version allows maximum 2 multi-user concurrent connections and up to 5 threads for database opened in Exclusive mode.

Multi-thread access is the case when there are several threads started in the single process and two or more of them can modify same tables simultaneously. This situation leads to lost changes or data corruption in database systems that are not multi-thread safe. All versions and editions of Accuracer are multi-thread safe. Look at Multi-Thread demo for example.

Multi-user access is when some different applications or multiple instances of the same application modifies the same tables simultaneously. There is no difference if these applications are run from different machines or they are started from the same computer - in any case it is a mult-thread access to the database. This situation is similar to multi-thread meaning that if database system is not multi-user there will be data losses or database corruption. The editions of Accuracer that does not support multi-user access are **SU Std** and **SU Pro**. When you use **SU edition** it is highly recomended to open database files in Exclusive mode (set property Exclusive of TACRDatabase component to True). This will not influences multi-thread support, but will prevent database file from corruption by other applications. All other versions, including **Trial** version supports multi-user access.

Each thread or application should use its unique **Session** for correct work. Session in Accuracer is identified by unique **SessionID** value that gets each TACRQuery or TACRTable component that connects to the database. Each TACRSession or TACRDatabase component gets SessionID value when it connects to the database file first time. All TACRTable and TACRQuery components linked to the corresponding TACRSession or TACRDatabase component uses this value. All modifications made by transaction 65 in the session are visible only to components of this Session until commit. Each session can contain only single active transaction.

Using Accuracer in Multi-Thread environment

There are two main methods of correct using Accuracer in multi-thread environment:
1) Create a TACRSession component in each thread and use same TACRDatabase component for all threads
2) Create a TACRDatabase component in each thread and use it by all TACRQuery and TACRTable components of this thread.
See Multi-Thread demo as an example.

Using Accuracer in Multi-User environment

There is no need in any configuration. Accuracer by default is ready for use in multi-user environment.
Read Locking Mechanism 64 topic for better understanding of locking model used in Accuracer.
You can run two instances of ACRManager or SQLConsole utility and open same table in both ot them to see how it works.
Note: Do not try to use **SU** version in multi-user mode !!!

Performance Optimization

Use Exclusive acces always when it is possible. If you need only multi-thread access to the database, but do not need multi-user access, set Exscluive property of TACRDatabase component to True before connecting the database. In this case all locks will be performed in memory, without locking any bytes in the physical database file. Locks will be checked much faster in this case.

If some tables can be used exclusively, set Exclusive property of TACRTable components to True

before opening them. This will essentially speed up access to the records.

Use transactions for speeding up data read and modification.

1.8.2 Locking Mechanism

Introduction

Accuracer can be used both in multi-user and multi-thread 62 environments.
Locking Mechanism is used for synchronization between differnent users or threads that modifies same database objects simultaneously. Locking protects data from reading or modification by other users or theads.

There are following locking objects in Accuracer:
- Low-Level database objects (Free Space Manager and Tables List) - used internally by Accuracer database engine.
- High-Level database objects (Tables and Records) - used by both Accuracer database engine and end-users.

Free Space Manager is a system that adds and removes pages (for later reuse) to the database file.
Table List is a system that handles tables stored inside the database file - allows to find, create, delete and rename tables.

Locking Mechanism (Locks Manager) implemented in accuracer performs all required locks automatically. There are two modes of the Locks Manager: InMemory and Disk modes.

InMemory mode is used when database file is opened Exclusively, that prevents it from modification by other applications.
Each session connected to the database file gets its unique identifier called SessionID. Different sessions are treated as different users, so modifications made by the transaction 65 in one session cannot be viewed by other sessions.
Also each session before locking table or record checks this lock type for compatibility with locks set by other session.
Single user version of Accuracer always uses InMemory mode. Locks Manager in InMemory mode does not locks bytes in the database file and works much faster than Disk mode.

Disk mode is designed for multi-user access to the database file (by different applications). In this mode Locks Manager performs locking and unlocking of physical bytes in the database file. There are reserved pages for the database and for each table for locking. The space required for Locks Manager is 1 byte for each connection to the database file and 11 bytes for each connection to each table. So maximum number of connectons (TACRDatabase.Options.MaxSessionCount) influences both size of the database file and performance of multi-user access to this file. If you set greater number of maximum concurrent connections each lock operation will work slower and vice versa. This approach allows to get high performance for locks checking and use Accuracer engine under different platforms. Most of all modern operating systems allows to lock and unlock bytes inside the database file, while other techniques may have problems under different platforms. For example, locking of bytes beyound the file works in all versions of MS Windows, but never work under Unix, Linux and Novell.

Lock modes

There are following lock types in Accuracer:
- IS - used for opening a table in shared mode, remains until close
- X - used for opening a table in exclusive mode, remains until close
- S - used for shortly locking the table in Read Only mode

- IRW - used for continious locking the table in ReadOnly mode
- RW - used for shortly locking the table in Exclusvie mode
- U - used for continious or shortly locking records (Edit, Delete operation in TACRDataset)
Each Session can lock only single record in the table. If one session locks record (U mode), other sessions cannot lock this record and cannot edit or update it, however they can read this record.

Before locking table or record Locks Manager checks new lock type for compatibility with locks set by other sessions.
If lock fails this session waits a timeout sepcefied in TACRDatabase.LockParams.Delay and tries to set lock again.
If number of unsuccessfully retries exceeds the value specified in TACRDatabase.LockParams.RetryCount an exception will be raised.

Table locks comaptibility schema:

Lock Mode	X	IS	S	IRW	RW
X	NO	NO	NO	NO	NO
IS	NO	YES	YES	YES	YES
S	NO	YES	YES	YES	NO
IRW	NO	YES	YES	NO	NO
RW	NO	YES	NO	NO	NO

Performance Optimization

There are following methods of improving multi-user access performance:
1) Use transactions 65
2) Open tables in Exclusive mode for critical operations
3) Combine methods 1 and 2

If you do not need multi-user access you can also open the whole database in Exclusive mode.

1.8.3 Transactions

Introduction

Transaction is the logical sequence of the database modification operations that can be treated as an atomic unit of work. Transactions have the following properties (ACID):

Atomicity
A transaction allows for the grouping of one or more changes to tables and rows in the database to form an atomic or indivisible operation. That is, either all of the changes occur or none of them do. If for any reason the transaction cannot be completed, everything this transaction changed can be restored to the state it was in prior to the start of the transaction via a rollback operation.

Consistency
Transactions always operate on a consistent view of the data and when they end always leave the data in a consistent state. Data may be said to be consistent as long as it conforms to a set of invariants, such as no two rows in the customer table have the same customer id and all orders have an associated customer row. While a transaction executes these invariants may be violated, but no other transaction will be allowed to see these inconsistencies, and all such inconsistencies will have been eliminated by the time the transaction ends.

Isolation

To a given transaction, it should appear as though it is running all by itself on the database. The effects of concurrently running transactions are invisible to this transaction, and the effects of this transaction are invisible to others until the transaction is committed.

Durability
Once a transaction is committed, its effects are guaranteed to persist even in the event of subsequent system failures. Until the transaction commits, not only are any changes made by that transaction not durable, but are guaranteed not to persist in the face of a system failure, as crash recovery will rollback their effects.

Transactions implementation in Accuracer

Accuracer supports transactions only for disk databases. In-Memory tables cannot be involved in the transaction.
All data modified by the transaction is stored in RAM, so if some failure will occurs duing the transaction processing all modifications will be lost and database will be in the same state as before starting the transaction. The database file can be corrupted only if failure occurs during the commit processing.

Multiple transactions on the same database file can be performed simultaneosly only if they are created in different sessions.
See Multi-User and Multi-Thread Support 62 topic to learn more about sessions in Accuracer.
All modifications made by the transaction cannot be accessed by other sessions until commit will be finished.

Transaction can be finished by performing Commit or Rollback. The Commit tries to write all changes made by the transaction to the database file and after that unlocks all tables involved in the transaction.
The Commit by default flushes file buffers after writing changes, so all data will be saved to the file immediately.
Optionally Commit can skip this process that works much faster than with flushing.

The Rollback discards all changes, removes all pages added during the transaction and after that unlocks all tables involved in the transaction.

All tables in the database opened before starting the tranasction or during its processing are automatically becomes involved in the transaction. It means that they are locked in S mode (See LockingMechansim 64 topic) and cannot be modified by other sessions.
If the transaction modifies some table it lock this table in IRW mode that means that other sessions cannot start Insert, Delete or Edit operations on this table and SQL statements INSERT, UPDATE, DELETE.
During the commit a transaction tries to set RW lock to all modified tables, and raises an exception if failed, so other sessions cannot read records from these tables.

Isolation level

The only isolation level in Accuracer is READ COMMITTED. It means that all changes made by the transaction cannot be viewed by other sessions until commit will be finished.

Executing a transaction

A transaction can be executed in two ways:
1) Using TACRDatabase component - methods StartTransaction, Commit, Rollback
2) Executing SQL statements START TRANSACTION, COMMIT, ROLLBACK

Transactions demo shows both of these methods.

Do not forget about handling an exceptions during the transaction processing: you should run Rollback manually if execption will be raised. Exceptions can be caused by impossibility to lock tables or by other reasons like constraints violation.

How the transactions increases the performance

The transaction locks all tables opened by current session and keeps these locks until Commit or Rollback will be called. Thus each table involved in the transaction cannot be modified by other sessions, so there is no need to re-read data from the database file and all changes are saved only during the Commit process, not after each single operation.

The maximum performance can be achieved by opening tables in Exclusive mode and running a transaction.
Even reading records works much faster inside the transaction.

Operations incompatible with transactions

All operations that requires Exclsuive access to the table or database cannot be performed when transaction is started.
However, all tables that are not invloved in the transaction cannot be accessed in Exclusive mode.

Here is a list of operations that are incompatible with transactions:
- Repairing database, Compacting database or Chaning database settings
- Emptying, Restructuring, Deleting and Renaming tables involved in the transaction
- Creating and dropping indexes on tables involved in the transaction

1.9 Client-Server Engine

1.9.1 Introduction

Introduction

Accuracer supports both Client-Server and File-Server (Multi-User 62) technologies.
When you use File-Server mode the database is specified by the DatabaseFileName property of TACRDatabase component (LocalDatabase property should be set to true). In this case each client application reads and writes to the database file directly, using OS locks for synchronization.
When you use Client-Server mode the database is specified by ConnectionParams property of TACRDatabase component (LocalDatabase property should be set to false). Client application sends requests to the database server via network using UDP-based protocol. Server application executes client requests and sends replies to them, accessing database files either exclusively (if OpenDatabasesInExclusiveMode is set to true) or as a File-Server.

How To Use

1) Using a database server.
You can use the Accuracer Database Server as a Windows NT / XP service as well as Windows application.
To install server as a service you should run it with /install option:
```
c:\Accuracer\Utils\Bin\ACRServer\AccuracerDatabaseServer.exe /install
```

After that you can start and stop it as a service.
To unistall service run the server with /unistall option:

c:\Accuracer\Utils\Bin\ACRServer\AccuracerDatabaseServer.exe /uninstall

Also you can run it as typical Windows application:
c:\Accuracer\Utils\Bin\ACRServer\AccuracerDatabaseServer.exe
In this case you can use a popup menu on tray icon to start and stop the server.

The default settings are applied when server is starting (setting Active to True) if there is no configuration file.
In thi case server will create the configuration file with default settings at first start.

Server provides access only to databases specified in properties DatabaseNames and DatabaseFileNames.
Default values for these properties points to Demos\Data\DBDemos.adb database.

Server port can be set via LocalPort property of TACRServer.

To work with clients from remote machines you should set the IP-address of the server in the **LocalHost** parameter of the configuration file. Default value 'localhost' allows to work only with clients started on the same machine.

2) Connecting to the database on database server.
You can use a TACRDatabase component for connecting to a remote database.
Set LocalDatabase property to False, specifty necessary connection parameters via ConnectionParams property (DatabaseName, RemoteHost, RemotePort and LocalPort) and open database (Open or Active := True).

Default settings allows to test in on local machine:
DatabaseName = DBDemos
RemoteHost = localhost
RemotePort = 6669
LocalPort = 6668

See demo Client as an example of connecting to a remote database.

If OpenDatabasesInExclusiveMode is set to true the maximum number of connections to single database is:
- 5 in trial version;
- 2^31 in commercial version;

If OpenDatabasesInExclusiveMode is set to false the maximum number of connections to single database is:
- 2 in trial version;
- MaxSessionsCount parameter of the database file in commercial version;

Advanced Features

Accuracer Database Server provides you a possiblity to change any SQL query before execution or abort it.
Thus you can make your application more flexible:
- you can change data definition without recompiling and reinstalling client applications
- you can block some SQL queries for security reasons
- you can log all SQL queries executed by clients
For more information read TACRServer.OnSQL event description in Accuracer Component Reference.

Another great advantage of the Accuracer Server is the custom messages support.

Now you can make any communication between server and any client connected to it.
You can send and receive text, binary and stream messages by both sides: client and server.
Thus you can communicate any client with another client (through the server), client with server and server with client.
And these messages can be sent and received at any time, simultaneously with accessing the tables and executing SQL scripts.
For more information look at events OnReceiveTextMessage, OnReceiveBinaryMessage, OnReceiveStreamMessage and method SendMessage of TACRDatabase and TACRServer components in Accuracer Component Reference.

Accuracer can compress and / or encrypt network traffic (ConnectionParams property of TACRDatabase and CryptoParams property of TACRServer).

Limitations

There are following limitations of the current Client-Server version:
- RepairTable, FlushFileBuffers, GetLastAutoinc and SetLastAutoinc does not supported
- OnFilterRecord does not work with remote table and live remote queries

We will try to remove most of them in next version (excluding OnFilterRecord).

1.10 Migration

1.10.1 Overview

Introduction

Now there are lots of various database systems for different platforms in the world.
In most cases we try to use our favorite database for all our projects, but sometimes we faces with the fact that it cannot suit our needs anymore. There are lots of reasons for that like lack of features, technical support or documentation, unreasonable cost, low performance, too much resource usage (RAM, CPU, disk space or just too large number of files), security holes, moving to another platofrm, etc.

This is a good time for migartion to other database system like our **Accuracer**.

How To Start

There are two main steps for migration:
1) Moving data to Accuracer
2) Updating projects and applications

We recommend to start with first one, as you will be able to test new database, examine the performance, check the database file compactness, make sure that all required functionality is supported, look at new feature set than can be used now before starting more complex part of this job - step 2.

We know that there are lots of database systems and development environments, so we will divide them on three separate groups:
1) BDE or ODBC 70
2) EasyTable 70
3) Other database systems and platforms 70

Note: If you have any problems during the migration process, contact our Technical Support Team 5 - we will help as soon as possible.

1.10.2 Migration from BDE

Introduction

The migration process is very simple:
1) Convert 71 your data by DBTransfer Utility
2) Replace your TTable, TQuery, TSession and TDatabase comonents to TACRTable, TACRQuery, TACRSession and TACRDatabase components
3) Set DatabaseFileName property of TACRDatabase components to the path to database file instead of Alias / Directory.
4) Set Exclusive properties of TACRTable components to the same values as in TTable components
5) If you need to access database file only from single instance of the application (single-user or multi-thread access 62), set Exclusive property of TACRDatabase to True.
6) Remove DBTables unit from uses clauses of your units. Otherwise your project will still require BDE.

There is no need in any other configurations.

1.10.3 Migration from EasyTable

Introduction

The migration process is very simple:
1) Convert 71 your data by Convert Utility
2) Replace your TEasyTable, TEasyQuery, TEasySession and TEasyDatabase comonents to TACRTable, TACRQuery, TACRSession and TACRDatabase components
3) Set DatabaseFileName property of TACRDatabase components to the path to Accuracer database file.
4) Set Exclusive properties of TACRTable components to True for tables that should be accessed exclusively.
5) If you need to access database file only from single instance of the application (single-user or multi-thread access 62), set Exclusive property of TACRDatabase to True.
6) Remove EasyTable and Etbl* units from uses clauses of your units. Otherwise your project will still require EasyTable.

There is no need in any other configurations.

1.10.4 Migration from other database systems and platforms

Introduction

Read Import and Export 71 topic to learn how you can move your data to Accuracer.
If your project uses database that provides TTable, TQuery, TSession and TDatabase - analogical components then migration process is similar to migratiing from BDE 70. Replacing ADO, Interbase or dbExpress components is almost the same task.

In other cases you have to rewrite all your code to use Accuracer components.

Note: If you have any problems during the migration process, contact our Technical Support Team - we will help as soon as possible.

1.10.5 Import and Export

Introduction

There are two methods of importing data to Accuracer:
1) Using Accuracer utilities DBTransfer or Convert
2) Making your own converter

The first method can be used if your database can be accessed via BDE or ODBC (like Paradox, Interbase, Access, DBase and FoxPro, Oracle, SQLServer). Otherwise you can easily make your own converter.

Importing tables from BDE or ODBC

If you need to move data from the database that can be accessed via BDE or ODBC the better way is to use DBTransfer utility located in <Accuracer installation folder>\Utils\Bin\DBTransfer\. You can select any existing Paradox or DBase (FoxPro) table and import it by the DBTransfer utility.
Also you can create any BDE or ODBC alias, run DBTransfer and import any tables from this alias.

Importing tables from EasyTable

Just run Convert utility and select existing EasyTable database file.

Importing tables from MySQL

See demo MySQLImport.

Importing tables from CSV (coma-separated values)

See demo CSVImport.

Importing tables from other database systems

If your database cannot be accessed via BDE or ODBC and it have no export utilities that can covert it to one of BDE-compatible formats then you have to make your own converter.

It is thery easy task if you can access the tables in Delphi, C++ Builder or Kylix:
- Create new application
- Create all necessary components for accessing your database
- Create TACRDatabase and TACRTable component, set Database name property of both components to the same value, like 'TestDB'
- Set DatabaseFileName property of TACRDatabase component to the existing database file, like 'c:\test.adb'. You can create database using either ACRManager utility or CreateDatabase method of TACRDatabase
- Create new converter procedure - open existing table in your database by table component

(derived from TDataset), close Accuracer table (TACRTable.Close), set table name (TACRTable.TableName) and call import method of TACRTable with your table component (TACRTable.ImportTable)
See the Convert utility source code as an example.
<Accuracer installation folder>\Utils\Source\Convert\

If your database cannot be accessed by TDataset descendant component you can try to export it to SQL script or contact our Support Team 5.

Note: If you have any problems during the migration process, contact our Technical Support Team 5 - we will help as soon as possible.

1.11 Tuning and Optimizations

1.11.1 Overview

Introduction

Modern databases, like Accuracer, provides lots of different settings that can be used for improving performance, making database file more compact, using less amount of RAM, etc. If you do not sure what do you need please, contact our Technical Support Team 5.

Using Indexes

If you perform searching or filtering on large tables the main way of improving performance is to create necessary indexes.
It can be done by using TACRTable.IndexDefs property before creating a table or running TACRTable.AddIndex method or CREATE INDEX 60 SQL Statement when table exists.

If you perform search on the single field ('Name' for example) you should create index on this field.
It is very important to create index on string fields (char, varchar, wide char or wide varchar) with same case sensitivity setting like in searching. If your perform case-insensitive search you should create case-insensitive index, otherwise create case-sensitive index.

Example 1:

```
TACRTable1.AddIndex('case_ins_index', 'Company', [ixCaseInsensitive]);
TACRTable1.Locate('Company', ['AidAim Software'], [loCaseInsensitive]);
```

You should create multiple-field index if you perform search with conditions for multiple fields.

Example 2:

```
TACRTable1.AddIndex('complex_index', 'Company;Contact;Phone', []);
TACRTable1.Locate('Company;Contact;Phone', VarArrayOf(['Sight Diver','P']), loPartialKey);
```

SELECT statements also runs faster when all necessary indexes are exists.
We recommend to create indexes for each field that is used in DISTINCT, WHERE, ORDER BY and FROM clauses.

If you use ORDER BY with multiple fields, create a single index on all these fields like in Example 2.

If you use context search with LIKE operator you should not create any indexes - it will not improve preformance.

Note: Indexes decreases speed of table modification, especially with Primary or Unique options.

Optimizing Navigation

You can use TACRDataset methods LockTable / UnlockTable for optimizing speed of loading records.
Example:

```
ACRTable1.LockTable;
try
  ACRTable1.First;
  while not ACRTable1.Eof do
   begin
    // ... some record processing
    ACRTable1.Next;
   end;
finally
  ACRTable1.UnockTable;
end;
```

Exclusive Access

If you do not need multi-user access 62, set Exclusive property of TACRDatabase or TACRTable components to True. This will significantly increase the performance, but other applications will not be able to access the tables and databases opened exclusively.

Maximum Number of Connections

You can set the maximum number of connections for each database (using ACRManager or TACRDatabase.Options property) to any value from 1 to ACRMaxSessionCount constant. Smaller value will increase performance in multi-user environment and also provide more compact database file.

How To Make Your Database File More Compact

You can make your file more compact if you will use varchar fields instead of long fixed length string fields.
Also you can use compression for BLOB and Varchar fields.
Read BLOB and Varchar fields 14 topic for more details.
Another tip is to run TACRDatabase.CompactDatabase method periodically (or use ARCManager utility) for removing unused space inside the database file. You can use Density property of TACRDatabase component to get the percent of used space inside the database file.

Other Settings

See Variables and Constants topic in Reference Guide.

Note: If you have any problems with tuning and optimizations, contact our Technical Support Team 5 - we will help as soon as possible.

1.12 Appendix

1.12.1 Differences from BDE

Main differences:

- Supported data types 74
- Maximum fields per table: ~ 2^31
- Maximum indexes per table: ~ 2^31
- Maximum index fields per index: ~ 2^31
- Maximum field name: 255 characters
- BLOB and Varchar compression algorithms: ZLIB, BZIP, PPM
- Sequences support (AutoInc fields based on sequences)
- Restructure table
- Unicode support
- No external drivers or dlls required
- Kylix version is available
- Strong database encryption with lots of algorithms and settings
- ftString field type is a Varchar equivalent - variable length string field
- ftFixedChar field type is a Char equivalent - fixed length string field

If you wish to inform us which features you need first of all, please, be sure to contact us: support@aidaim.com.

If you want to be informed about new releases of this component, you can also subscribe to our news list: http://www.aidaim.com/info/subscr.php

1.12.2 Supported data types

Accuracer supports the following data types of fields in tables:

SQL data type	Description	Corresponding TFieldType
Char, FixedChar	Fixed character field	ftFixedChar
Varchar, Varchar2, String	Character or variable length string field	ftString
WideChar, FixedWideChar	Fixed wide character field	ftWideString
WideVarchar, WideString	Wide character or variable length wide string field	ftWideString
Shortint, SignedInt8	8-bit integer field	ftSmallint
Smallint, SignedInt16	16-bit integer field	ftSmallint
Integer, SignedInt32	32-bit integer field	ftInteger
Largeint, Int64, SignedInt64	64-bit integer field	ftLargeint
Byte, UnsignedInt8	Byte field	ftWord
Word, UnsignedInt16	16-bit unsigned integer field	ftWord
Cardinal, UnsignedInt32	32-bit unsigned integer field	ftLargeint

AutoInc, AutoIncInteger	Auto-incrementing 32-bit integer counter field	ftAutoinc
AutoIncShortint	Auto-incrementing 8-bit integer counter field	ftAutoinc
AutoIncSmallint	Auto-incrementing 16-bit integer counter field	ftAutoinc
AutoIncLargeint	Auto-incrementing 64-bit integer counter field	ftAutoinc
AutoIncByte	Auto-incrementing byte counter field	ftAutoinc
AutoIncWord	Auto-incrementing 16-bit unsigned integer counter field	ftAutoinc
AutoIncCardinal	Auto-incrementing 32-bit unsigned integer counter field	ftAutoinc
Single	Single floating-point numeric field	ftFloat
Float, Double	Double floating-point numeric field	ftFloat
Extended	Extended floating-point numeric field	ftFloat
Boolean, Logical, Bool, Bit	Boolean field	ftBoolean
Currency, Money	Money field	ftCurrency
Date	Date field	ftDate
Time	Time field	ftTime
DateTime	Date and time field	ftDateTime
TimeStamp	Date and time field accessed through dbExpress	ftTimeStamp
Bytes	Fixed number of bytes (binary storage)	ftBytes
VarBytes	Variable number of bytes (binary storage)	ftVarBytes
Blob	Binary Large OBject field	ftBlob
Graphic	Bitmap field	ftGraphic
Memo, Clob	Text memo field	ftMemo
FormattedMemo, FmtMemo	Formatted text memo field	ftFmtMemo
WideMemo, WideClob	Unicode text memo field	ftMemo

1.12.3 Internationalization and localization

Introduction

This topic discusses guidelines for writing applications you plan to distribute to an international market. By means of the ahead planning, you may reduce the amount of code and time necessary to make your application operate in its foreign market as well, as it does in its domestic market.

Some things you should know for writing international applications with Accuracer.

Currency, float numbers and date/time format

Sometimes you have to convert Currency, Date/Time or float value to string. For example if you need to set Filter property of Accuracer for condition like 'Birthday=01/01/1970'.
Accuracer always uses current values of DateSeparator, TimeSeparator and DecimalSeparator. So you should use DateToStr/TimeToStr or FloatToStr functions to get converted date/time or float value.

Locale and strings sort order

You may use the IndexName and IndexFieldNames properties to set the current index order, and consequently, sort the current table based upon the index definition for the selected index order. However sorting order for strings depends on current system/user locale and it is specific for various languages.
Accuracer uses locale specific string operations, so if you use ftString data type then all records being sorted by this field will be sorted using current system locale.
If you want to support Asian languages you should use Unicode character set.

Unicode support

In the Unicode character set, each character is represented by two bytes. Thus a Unicode string is a sequence of two-byte words, not individual bytes. Unicode characters and strings are also called wide characters and wide character strings. The first 256 Unicode characters map to the ANSI character set.
Accuracer implements Unicode support through the ftWideString data type.

The following example shows how to set and get data of Unicode field.

```
var ws: WideString;
 with MyAccuracer do
  begin
   // get data from Unicode field
   ws := FieldByName('Unicode').Value;
   // do something with data in ws
   ws := 'example string';
   // set data to Unicode field
   Insert;
   FieldByName('Unicode').Value := ws;
   Post;
  end;
```

WideMemo fields can be accessed using TACRDataset methods SetWideMemoField and GetWideMemoField:

```
var ws: WideString;
 with MyAccuracer do
  begin
   // do something with data in ws
   ws := 'example string';
   // set data to Unicode field
   Insert;
   SetWideMemoField(FieldByName('Unicode'), ws);
   Post;
   // get data from Unicode field
   ws := GetWideMemoField(FieldByName('Unicode'));
  end;
```

Note: Wide string fields can be used only with Windows versions that supports CompareStringW

function (NT, 2000, XP, 2003). Windows 9x, Me does not support wide strings.

1.12.4 Limitations

- WideString field type is not supported in Delphi 4, C++ Builder 4
- SQLTimeStamp field is not supported in Delphi 4,5, C++ Builder 4,5
- BCD fields are not supported
- Trial Version can execute only SELECT SQL statements
- Maximum number of multi-user connections in Trial Version is 2
- Maximum number of multi-thread connections in single-user (Exclusive mode) in TrialVersion is 5
- SU Std Version can be used only in single-user enironment or in multiple threads inside the single instance of application.
- OnFilterRecord does not work with remote table and live remote queries
- RepairTable, GetLastAutoinc and SetLastAutoinc does not supported in remote databases

Index

- A -

- B -

- C -

- D -

- E -

- F -

- G -

- H -

- T -

- U -

- V -

- W -

- Y -

Endnotes 2... (after index)

Back Cover