Microolap DAC for MySQL

Programmer's reference

Chernogolovka, Russia
2021

Contents

1. Welcome!

Thank you for your interest in Direct Access Components for MySQL!

DAC for MySQL is a member of Microolap Direct Access Components line of products.

Our DAC products allow to create Delphi/C++Builder applications with the direct access to SQL servers without BDE, ODBC and even without *libmysql.dll*.

DAC for MySQL supports all available platforms in this IDE's:

- Delphi 5-7,
- C++Builder 5, 6;
- Borland Developer Studio 2005-2006, Turbo Delphi 2006, Turbo C++ 2006;
- CodeGear RAD Studio 2007 (both Delphi 2007 and C++Builder 2007);
- CodeGear Delphi and C++Builder 2009 (Tiburon);
- Embarcadero RAD Studio 2010 (both Delphi 2010 and C++Builder 2010);
- Embarcadero RAD Studio XE (Delphi XE and C++Builder XE);
- Embarcadero RAD Studio XE2 (Delphi XE2 and C++Builder XE2);
- Embarcadero RAD Studio XE3 (Delphi XE3 and C++Builder XE3);
- Embarcadero RAD Studio XE4 (Delphi XE4 and C++Builder XE4);
- Embarcadero RAD Studio XE5 (Delphi XE5 and C++Builder XE5);
- Embarcadero RAD Studio XE6 (Delphi XE6 and C++Builder XE6);
- Embarcadero RAD Studio XE7 (Delphi XE7 and C++Builder XE7);
- Embarcadero RAD Studio XE8 (Delphi XE8 and C++Builder XE8);
- Embarcadero RAD Studio 10 Seattle (Delphi 10 Seattle and C++Builder 10 Seattle).
- Embarcadero RAD Studio 10.1 Berlin (Delphi 10.1 Berlin and C++Builder 10.1 Berlin).
- Embarcadero RAD Studio 10.2 Tokyo (Delphi 10.2 Tokyo and C++Builder 10.2 Tokyo).
- Embarcadero RAD Studio 10.3 Rio (Delphi 10.3 Rio and C++Builder 10.3 Rio).
- Embarcadero RAD Studio 10.4 Sydney (Delphi 10.3 Sydney and C++Builder 10.3 Sydney).
- Embarcadero RAD Studio 11 Alexandria (Delphi 11 Alexandria and C++Builder 11 Alexandria).

All these products were developed according to the following requirements:

1. Avoid the weaknesses of BDE/ODBC technology:

Deployment:

You add to your distributive more than 5Mb with BDE;
If another BDE-based program will be installed on user's workstation it can conflict with your application;
ODBC profile on user's workstation can be changed or damaged;
Possible ODBC drivers conflicts.

Performance:

BDE uses another layer of middleware (ODBC). So, in this case you have two resource eaters: BDE and ODBC.

BLOB fields:
Several of the SQL Links drivers provided with Delphi Enterprise Edition do not work with BLOB data types correctly;

Data access:
BDE cannot access some RDBMS data types at all;

Cost:
Only the Enterprise Edition of Delphi (~$3000) includes the BDE SQL Links required for connecting to DB servers.
DAC: Ability to work with Professional Edition.

2. Keep migration of old BDE/ODBC based projects to DAC easy:

No additional knowledge required:
DAC components are **TDataSet** compatible. Implementing of additional RDBMS-specific functions and properties must be obvious;

No external modules:
DAC for MySQL does not require any additional external modules from MySQL (even *libmysql.dll*).

See also: Components list

1.1. Installation

Installation:

- Unzip archive file to any location you prefer;
- Run .msi file and follow installation application instructions.

There are only compiled binaries installed for Trial package.

If you have one of Personal, Business, Commercial or Educational license packages you can choose to install or not install binaries during installation process.

After installation application completes DAC for MySQL tab at the Components palette will appear.

DAC for MySQL With Sources version

- Start your Delphi (if you're using Rad Studio for both programming languages you need start Rad Studio);
- Choose Main menu File/Open, and then select directory in which DAC for MySQL was installed;
- Open MySQLDACXX.dproj, where XX - is the version of your IDE.

- Select the target platform which you need in the Project Manager window and Build MySQLDACXX.dproj. You can build MySQLDACXX.dproj for all platforms, but build for Win32 is the required condition.
- Close MySQLDACXX.dproj without save.
- Open dcl_MySQLDACXX.dproj. Build & Install it. This is design time package so you need build it only for Win32 target platform.
- Close dcl_MySQLDACXX.dproj without save.
- Add path to the DAC for MySQL sources for desired platform.

DAC for MySQL With Sources version (if you have C++ Builder only)

- Start your C++ Builder;
- Choose Main menu File/Open, and then select directory in which DAC for MySQL was installed;
- Open MySQLDACXXCB.cbproj, where XX - is the version of your IDE.
- Select the target platform which you need in the Project Manager window and Build MySQLDACXXCB.cbproj.
- Once you have built platforms you need, build package for Win32 platform. This is REQUIRED condition!
- Close MySQLDACXXCB.cbproj without save.
- Open dcl_MySQLDACXXCB.cbproj. Build & Install it. This is design time package, so you need build it only for Win32 target platform.
- Close dcl_MySQLDACXXCB.cbproj without save.
- Add path to the DAC for MySQL sources for desired platform.

Installation for Lazarus

- Start your Lazarus IDE;
- Choose Main menu **Open**, and then select DAC for MySQL sources directory;
- Open *MySQLDACL.lpk*.
- Click the **Open Package** button on dialog message.
- Select **Use>> Install**.
- Confirm rebuilding Lazarus.
- Click **Ignore All** button if *"ambiguous unit found"* message dialog appears.
- Set path to the DAC for MySQL sources in the *FPC.cfg* file.

Notes

For working with **MySQL 8** the OpenSSL libraries should be present. You can add path to these libraries by PATH variable or copying to the project folder.

For using DAC for MySQL in Lazarus on Windows you need zlib1.dll.

1.2. Registration

Thank you for your interest in purchasing of DAC for MySQL!

You can choose licensing options and online secure services on DAC for MySQL page.

DAC for MySQL is a royalty-free product. This means, you have to register DAC for MySQL for each developer, but not for each user of application you have developed!

After purchase you will receive an email with registered version download information in 1-2 business days or even earlier.

Support for registered users:

- Private support account in ticket support system. You can create new support ticket here: http://www.microolap.com/support/ticket_edit.php;
- Support by e-mail;

Update policy:

- Any updates during one year since purchasing for free;
- Update subscription renewals with significant discount.

1.3. Components list

There are DAC for MySQL components in alphabetical order with short descriptions.

Icon	Component	Description
	TBDE2MySQLDAC	This component is intended for the conversion of BDE Database objects into DAC for MySQL Database objects. It provides an easy way of migration from BDE components to DAC for MySQL.
	TMySQLBatchExecute	This component can execute SQL scripts containing more than one SQL statement.
	TMySQLDatabase	Provides discrete control over a connection to a single database

		in a database application.
	TMySQLDirectQuery	Component for high-speed (3-4 times faster then with TMySQLQuery component) data fetching. It is not TDataset compatible.
	TMySQLDump	Allows to get SQL script with a dump of a Database. This script can be executed on another MySQL server by TMySQLBatchExecute component.
	TMySQLMacroQuery	TMySQLMacroQuery is the descendant of TMySQLQuery component and supports all of its properties, methods, events, and functionalities. The difference is in Macros and MacroChar properties which help to modify SQL script text in design-time and run-time with easy.
	TMySQLMonitor	Monitors dynamic SQL passed to the MySQL server.
	TMySQLQuery	Encapsulates a dataset with a result set that is based on an SQL statement.
	TMySQLStoredProc	Provides full support for MySQL 5.0+ stored procedures.
	TMySQLTable	Encapsulates a database table.
	TMySQLTools	Allows to run MySQL diagnostic and repair operations such as Repair, Check, Analyze, Optimize, Backup and Restore.
	TMySQLUpdateSQL	Applies updates on behalf of queries or stored procedures that can't post updates directly.

2. TBDE2MySQLDAC

 This component is intended for the conversion of BDE, ZeosDB, dbExpress, ADO Database objects into DAC for MySQL Database objects. It provides an easy way of migration from BDE, ZeosDB, dbExpress, ADO components to DAC for MySQL.

BDE component	ZeosDB component	dbExpress component	ADO component	DAC for MySQL analog
TTable	TZTable	TSQLTable	TADOTable	TMySQLTable
TQuery	TZQuery	TSQLQuery	TADOQuery	TMySQLQuery
TStoredProc	TZStoredProc	TSQLStoredProc	TADOStoredProc	TMySQLStoredProc
-	-	TSQLDataset	TADODataset	TMySQLQuery

See also: Properties

2.1. Properties

Please see TBDE2MySQLDAC properties short descriptions below:

ConvertComponents
Sets which one of BDE, ZeosDB, dbExpress, ADO components must be converted to DAC for MySQL analogs.

Database
Points to TMySQLDatabase component which sets a DB to be connected with.

DeleteSourceComponents
Sets that all BDE, ZeosDB, dbExpress, ADO components must be deleted when DAC for MySQL ones are created.

Execute
Performs the BDE, ZeosDB, dbExpress, ADO to DAC for MySQL conversion process.

2.1.1. ConvertComponents

Since v2.6.0

Sets which one of BDE, ZeosDB, dbExpress, ADO components must be converted to DAC for MySQL analogs.

Syntax:

```
type
  TConvertComponent = (convBDE, convADO, convDBX, convZeos);
  TConvertComponents = set of TConvertComponent;
property ConvertComponents: TConvertComponents;
```

Description:

Include corresponding values to **ConvertComponents** set to convert certain components to DAC for MySQL analogs.

Set member	Converted components
convBDE	BDE database components
convADO	ADO database components
convDBX	DBX database components
convZeos	ZeosDBO database components

2.1.2. Database

Points to TMySQLDatabase component which sets a DB to be connected with.

Syntax:

```
Database: TMySQLDatabase;
```

2.1.3. DeleteSourceComponents

Since v2.6.0

Sets that all BDE, ZeosDB, dbExpress, ADO components must be deleted when DAC for MySQL ones are created.

Syntax:

```
property DeleteSourceComponents: Boolean;
```

Description:

If set to **True** all **TTable**, **TQuery** and so on components will be deleted from project. In other case they will be left on the form or data module. However, data sources and fields descriptions will be detached from there and moved to the DAC for MySQL replacement.

> You may want to left BDE, ZeosDB, dbExpress, ADO components to check correctness of the migration process manually.

2.1.4. Execute

Performs the BDE, ZeosDB, dbExpress, ADO to DAC for MySQL conversion process.

Syntax:

```
property Execute: Boolean;
```

Description:

Click this item within the **Object Inspector** at design time to convert BDE, ZeosDB, dbExpress, ADO objects (**TQuery**, **TTable** and so on) into DAC for MySQL ones (TMySQLQuery, TMySQLTable and so on respectively).

3. TMySQLBatchExecute

 **TMySQLBatchExecute** component can execute SQL scripts containing more than one SQL statement. SQL script text must be set in SQL property. Each SQL statement must be separated by the symbol set in the Delimiter property (";" by default).

Database property sets TMySQLDatabase object connected to DB in which SQL script will be executed.

See also: Properties, Methods, Events

3.1. Properties

Please see TMySQLBatchExecute properties short descriptions below:

Aborted
Returns **True** if the script execution was aborted by AbortExecute method.

Database

Points to **TMySQLDatabase** component which sets DB to be connected.

SQL
Contains the text of the SQL script to be executed.

Delimiter
Sets the SQL statements delimiter.

Action
Sets actions when SQL script statements errors occur.

RowsAffected
Returns total number of records changed by executed SQL script.

StatementBeginLine, StatementEndLine, StatementBeginCharacter, StatementEndCharacter, StatementAbsoluteBeginCharacter, StatementAbsoluteEndCharacter
These properties contain information about position of current statement in the script to be executed.

StatementNumber
Returns current statement number during script execution.

3.1.1. Aborted

Returns **True** if the script execution was aborted by AbortExecute method.

Syntax:

```
property Aborted : boolean;
```

Description:

Examine **Aborted** property value after script execution to determine if the script execution was aborted by AbortExecute method call.

See also: AbortExecute method

3.1.2. Action

Action property sets actions when SQL script statements errors occur.

Syntax:

```
Action: TMySQLBatchAction;
Type
TMySQLBatchAction = (baFail, baAbort, baIgnore, baContinue);
```

Description:

Action property sets actions when SQL script statements errors occur.

baFail
Exception will be generated (default).

baAbort
Eabort exception will be called, script execution will be stopped, error message will be not displayed.

baIgnore
Error message will be not displayed, script execution will be continued from the next SQL statement.

baContinue
Error message will be displayed, script execution will be continued from the next SQL statement.

3.1.3. Database

Points to TMySQLDatabase component which sets DB to be connected.

Syntax:

```
Database: TMySQLDatabase
```

3.1.4. Delimiter

Sets the SQL statements delimiter.

Syntax:

```
Delimiter: string;
```

Description:

SQL statements included into SQL script set in SQL property must be separated by the string defined in **Delimiter** property ("**;**" by default).

3.1.5. RowsAffected

Returns total number of records changed by executed SQL script.

Syntax:

```
RowsAffected: LongInt;
```

Description:

In run-time **RowsAffected** returns total number of records changed by executed SQL script.

3.1.6. SQL

Contains the text of the SQL script to be executed.

Syntax:

```
SQL: TStringList;
```

Description:

SQL property contains the text of the SQL script which will be executed on. ExecSQL method call. Each SQL statement must be ended with a symbol set in Delimiter property.

3.1.7. Statement Position Properties

These properties contain information about position of current statement in the script to be executed.

Syntax:

```
property StatementAbsoluteBeginCharacter: Cardinal;
property StatementAbsoluteEndCharacter: Cardinal;
property StatementBeginCharacter : Cardinal;
property StatementBeginLine : Cardinal;
property StatementEndCharacter : Cardinal;
property StatementEndLine : Cardinal;
```

Description:

In run-time these properties contain the information about position of current statement in executed script. You can use these properties to highlight current statement in TMemo component in case of any error.

StatementBeginLine
The first line number of the statement, starts from **1**.

StatementEndLine
The last line number of the statement, started from **1**.

StatementBeginCharacter
Character number in **StatementBeginLine** string where current statement starts. Characters numbering is started from **1**.

StatementEndCharacter
Character number in **StatementEndLine** string where current statement ends. Characters numbering is started from **1**.

StatementAbsoluteBeginCharacter
Character number in whole SQL script where current statement begins. Characters numbering is started from **1**.

StatementAbsoluteEndCharacter
Character number in whole SQL script where current statement begins. Characters numbering is started from **1**.

3.1.8. StatementNumber

Returns current statement number during script execution

Syntax:

```
property StatementNumber : cardinal;
```

Description:

In run-time **StatementNumber** returns current statement number.

3.2. Methods

Please see TMySQLBatchExecute methods short descriptions below:

AbortExecute
Aborts script execution and returns from ExecSQL method.

ExecSQL
Executes SQL script set in SQL property.

3.2.1. AbortExecute

Aborts script execution and returns from ExecSQL method.

Syntax:

```
procedure AbortExecute;
```

Description:

Use **AbortExecute** method to abort script execution and return from ExecSQL method as soon as possible. You can call this method from one of components event handlers. Aborted property is set to **True** after this method call.

See also: ExecSQL method, Aborted property

3.2.2. ExecSQL

Executes SQL script set in SQL property.

Syntax:

```
procedure ExecSQL;
```

3.3. Events

Please see TMySQLBatchExecute events short descriptions below:

OnAfterStatement
Occurs immediately after execution of SQL statement from SQL script by ExecSQL method.

OnAfterExecute
Occurs immediately after SQL script was executed by ExecSQL method.

OnBeforeExecute
Occurs immediately before SQL script execution by ExecSQL method.

OnBatchError
Fires after an error occurs after execution of SQL statement from SQL script executed by ExecSQL method.

OnBatchErrorEx
Extended version of **OnBatchError** event. Fires after an error occurs after execution of SQL statement from SQL script executed by ExecSQL method.

OnProcessEx
Fires when component need to execute current statement from SQL script executed by ExecSQL method.

3.3.1. OnAfterExecute

OnAfterExecute event occurs immediately after SQL script was executed by ExecSQL method.

Syntax:

```
OnAfterExecute: TNotifyEvent;
```

3.3.2. OnAfterStatement

OnAfterStatement event fires immediately after execution of SQL statement from SQL script by ExecSQL method.

Syntax:

```
type
  TMySqlBatchAfterStatementEvent = procedure(Sender: TObject;
                                             SQLText: string;
  const StatementNo, RowsAffected: Integer;
  const Success: boolean) of object;
  OnAfterStatement: TMySqlBatchAfterStatementEvent;
```

Description:

Parameters:

SQLText
SQL query statement text;

StatementNo
The number of the SQL statement in SQL script.

RowsAffected
The quantity of records changed after query execution;

Success
Return a value of the boolean type with query execution result.

3.3.3. OnBatchError

Deprecated since v2.6.0, use OnBatchErrorEx

OnBatchError event fires after an error occurs after execution of SQL statement from SQL script executed by ExecSQL method.

Syntax:

```
OnBatchError: TMySQLBatchErrorEvent;
Type
TMySQLBatchErrorEvent = procedure(Sender: TObject;
                                      E: EMySQLDatabaseError;
                                SQLText: String;
                             StatementNo: Integer) of object;
```

Description:

Parameters:

Sender
Points to **TMySQLBatchExecute** component generated this error;

E
An instance of **EmySQLDatabaseError** object which contains this error info;

SQLText
Contains the text of the statement in which this error occurs;

StatementNo
The number of the SQL statement in SQL script.

See also: OnBatchErrorEx event

3.3.4. OnBatchErrorEx

Since v2.6.0

Extended version of **OnBatchError** event. **OnBatchErrorEx** event fires after an error occurs after execution of SQL statement from SQL script executed by ExecSQL method.

Syntax:

```
OnBatchErrorEx: TMySQLBatchErrorExEvent;
type
TMySQLBatchErrorExEvent = procedure(Sender : TObject;
                                        E : EMySQLDatabaseError;
                                  SQLText : String;
                               StatementNo : Integer;
                               var aAction : TMySQLBatchAction) of object;
```

Description:

Use this event if you want to add some custom error processing. For example to ask user if he wants to continue. You can set **aAction** param value to **baIgnore** if user's answer is YES or to **baAbort** otherwise.

> Take a look at Action property for possible **aAction** parameter values

Parameters:

Sender
Points to **TMySQLBatchExecute** component generated this error;

E
An instance of **EmySQLDatabaseError** object which contains this error info;

SQLText
Contains the text of the statement in which this error occurs;

StatementNo
The number of the SQL statement in SQL script.

aAction
Allows to override default error action defined with Action property for current statement.

See also: ExecSQL method, Action property

3.3.5. OnBeforeExecute

OnBeforeExecute event occurs immediately before SQL script execution by ExecSQL method.

Syntax:

```
OnBeforeExecute: TNotifyEvent;
```

3.3.6. OnBeforeStatement

OnBeforeStatement event fires immediately before execution of SQL statement from SQL script by ExecSQL method.

Syntax:

```
type
  TMySqlBatchBeforeStatementEvent = procedure(Sender: TObject;
                                              SQLText: string;
  const StatementNo: Integer;
          var Allow: boolean) of object;
  OnBeforeStatement: TMySqlBatchBeforeStatementEvent;
```

Description:

Parameters:

SQLText
SQL query statement text;

StatementNo
The number of the SQL statement in SQL script.

Allow
Sets if allow this statement execution (**True**) or no (**False**).

3.3.7. OnProcessEx

Since v2.6.0

OnProcessEx event fires when component need to execute current statement from SQL script executed by ExecSQL method.

Syntax:

```
property OnProcessEx: TMySQLBatchProcessExEvent;
type
TMySQLBatchProcessExEvent = procedure(Sender            : TObject;
                                      const SQLText     : String;
                                      StatementType     : TMySQLStatementType;
                                      const StatementNo : Integer;
                                      var Processed     : boolean) of object;
TMySQLStatementType = (mstCursor, mstNonCursor);
```

Description:

Use this event if you want to execute statement by user code with some custom processing. For example you can show resultset for cursor-returning query. Otherwise statement will be executed by component itself without any special processing.

OnProcessEx event differs from OnBeforeStatement event. You can cancel statement execution at all in OnBeforeStatement event. OnProcessEx and OnAfterStatement events are not fired if you'll set Allow parameter of OnBeforeStatement event to **False**. And **OnProcessEx** event is just a way for some custom processing of executed statement.

Parameters:

Sender
Points to **TMySQLBatchExecute** component generated this error;

SQLText
SQL query statement text

StatementType
Type of current statement. Possible values are:
- **mstCursor** - Statement that return some resultset. This are SELECT, EXPLAIN, SHOW and DESCRIBE statements for now.
- **mstNonCursor** - All other statements.

StatementNo
The number of the SQL statement in SQL script

Processed
Set this parameter value to **True** if you don't want for component execute statement by itself. For example if you've executed it by yourself in event handler.

See also: OnBeforeStatement and OnAfterStatement events

4. TMySQLDatabase

 **TMySQLDatabase** provides discrete control over a connection to a single database in a database application.

Use **TMySQLDatabase** when a database application requires any of the following control over a database connection:

- Persistent database connections;
- Customized database server logins;
- Transaction control;
- Single-value queries.

Please read this FAQ section if you want to use Unicode strings in your application: How to use Unicode data in my application?

See also: Properties, Methods, Events

4.1. Properties

Please see TMySQLDatabase properties short descriptions below:

Connected
Indicates whether or not a database connection is active.

ConnectionCharacterSet
Sets connection character set.

ConnectionCollation
Sets connection collation.

ConnectionTimeout
Specifies the time interval to awaiting for connection is established.

ConnectOptions
Sets DB connection parameters.

DatabaseName
Specifies the name of the database to associate with this database component.

DataSetCount
Indicates the number of active datasets associated with the connection component.

DatasetOptions
Returns or sets common properties for all datasets (tables, queries, stored procedures) attached this TMySQLDatabase component.

DataSets
Provides an indexed array of all active datasets for a database component.

DesignOptions
Returns or sets database properties to organize the component behavior at design-time.

Exclusive
Deprecated property, use MultiThreaded instead

Handle
Specifies the database handle.

HandleShared
Specifies whether or not to share a database handle.

Host
Sets HOST on which server is running.

InTransaction
Indicates whether a database transaction is in progress or not.

IsSSLUsed
Read this property value to ensure that SSL encryption is used for connection to MySQL server.

KeepConnection
Specifies whether an application remains connected to a database even if no datasets are open.

LastInsertID
Get last inserted value of AUTO_INCREMENT column from MySQL server

LoginPrompt
Specifies whether a dialog appears immediately before opening a new connection.

MaxAllowedPacketSize
Sets *'max_allowed_packet'* connection parameter value in megabytes.

MultiThreaded
Allows usage of MySQL connection from several threads.

Params
Contains database connection parameters for the MySQL server.

Port
Sets server's port.

ReadOnly
Specifies that the database connection provides read-only access.

ServerVersion
Specifies server version like integer number.

SSLProperties
Sets options for SSL connection (encrypted protocol).

TransIsolation
Specifies the transaction isolation level for transactions.

UserName
The user ID with which you log on to the database.

UserPassword
To provide a password for the connection.

Utf8Used
Read this property value to ensure that UTF8 character set is used in the connection to MySQL server. This property is read-only.

WarningsCount
Returns number of warnings issued by server for latest query.

4.1.1. Connected

Indicates whether or not a database connection is active.

Syntax:

```
property Connected: Boolean;
```

Description:

Set **Connected** to **True** to establish a database connection without opening a dataset.

Set **Connected** to **False** to close a database connection. An application can check **Connected** to determine the current status of a database connection. If **Connected** is **True**, the database connection is active; if **False**, and the KeepConnection property is also **False**, then the connection is inactive.

> Set KeepConnection to **True** to avoid having to login to the server each time a database connection is reopened.

4.1.2. ConnectionCharacterSet

> Since v2.6.1

Sets connection character set.

Syntax:

```
property ConnectionCharacterSet: String;
```

Description:

DAC for MySQL executes 'SET NAMES <character set name> COLLATE <collation name>' query immediately after connection to the database is established, if at least one of **ConnectionCharacterSet** or ConnectionCollation properties values is not set in empty string.

For details about character sets and collations in MySQL please refer to http://dev.mysql.com/doc/refman/5.0/en/charset.html

> Supported character sets list may be received by executing 'SHOW CHARACTER SET' query.

> Please read this FAQ section if you want to use Unicode strings in your application: How to use Unicode data in my application?

See also: ConnectionCollation, Utf8Used properties

4.1.3. ConnectionCollation

> Since v2.6.1

Sets connection collation.

Syntax:

```
property ConnectionCollation: String;
```

Description:

DAC for MySQL executes 'SET NAMES <character set name> COLLATE <collation name>' query immediately after connection to database is established, if at least one of ConnectionCharacterSet or **ConnectionCollation** properties values is not set in empty string.

For details about character sets and collations in MySQL please refer to http://dev.mysql.com/doc/refman/5.0/en/charset.html

> Every character set has its own set of collations.
> You can get supported collations list by executing SHOW COLLATION LIKE 'utf8%' query replacing utf8 with desired character set.

See also: ConnectionCharacterSet property

4.1.4. ConnectionTimeout

Specifies the time interval to awaiting for connection is established.

Syntax:

```
property ConnectionTimeout: cardinal;
```

Description:

Use **ConnectionTimeout** to specify the time interval in seconds before an attempt to make a connection is considered unsuccessful. The default value is 30 seconds.

If a connection is successfully established before expiration of the seconds specified, then **ConnectionTimeout** has no effect. If the specified time expires and a connection has not been successfully established, the attempt is terminated and an exception is raised.

See also: Connect and ConnectWithConnectionOptionsDialog methods, OnConnectionFailure event

4.1.5. ConnectOptions

Sets DB connection parameters.

Syntax:

```
property ConnectOptions: TConnectOptions;
type
TConnectOption = (coCompress,
                  coFoundRows,
                  coIgnoreSpaces,
                  coInteractive,
                  coNoSchema,
                  coODBC,
                  coSSL);
TConnectOptions = set of TConnectOption;
```

Description:

The value of **ConnectOptions** is usually empty set [], but it can be set to a combination of the following flags in very special circumstances:

coCompress
Use compression protocol.

coFoundRows
Return the number of found (matched) rows, not the number of affected rows.

coIgnoreSpaces
Allow spaces after function names. Makes all functions names reserved words.

coNoSchema
Don't allow the *db_name.tbl_name.col_name* syntax. This is intended for ODBC only: it causes the parser to generate an error if you use that syntax, that is useful for trapping bugs in some ODBC programs.
coODBC
The client is an ODBC client.
coInteractive
Allow *interactive_timeout* seconds (instead of *wait_timeout* seconds) of inactivity before closing the connection.

coSSL
Use SSL (encrypted protocol).

Example of usage:

Use the following code to enable SSL usage at run-time:

```
mySQLDatabase1.ConnectOptions := mySQLDatabase1.ConnectOptions + [coSSL];
```

Don't forget to add **MySQLTypes** to your uses section.

See also: SSLCert, SSLKey, IsSSLUsed properties

4.1.6. DatabaseName

Specifies the name of the database to associate with this database component.

Syntax:

```
property DatabaseName: String;
```

Description:

Use **DatabaseName** to specify the name of the database to use with a database component.

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

See also: Host, Port, UserName, UserPassword properties

4.1.7. DataSetCount

Indicates the number of active datasets associated with the connection component.

Syntax:

```
property DataSetCount: Integer;
```

Description:

Use **DataSetCount** to determine the number of datasets listed by the DataSets property. DataSets includes only active datasets; the value of **DataSetCount** changes when datasets are opened and closed. Use **DataSetCount** as an upper bound when interacting through the DataSets property.

See also: Example: DataSetCount,DataSets

4.1.8. DatasetOptions

> Since v2.7.4

Returns or sets common properties for all datasets (tables, queries, stored procedures) attached to this **TMySQLDatabase** component.

Syntax:

```
TMySQLDatasetOption = (mdsoZeroDateAsNull, mdsoFTStringAsVarchar,
mdsoCaseInsensitiveLocalSort);
TMySQLDatasetOptions = set of TMySQLDatasetOption;
property DatasetOptions: TMySQLDatasetOptions;
```

Description:

Set **DatasetOptions** to adjust behavior of all TMySQLDataset descendants attached to this TMySQLDatabase component. **DatasetOptions** is a set drawn from the following values:

mdsoZeroDateAsNull
Enables treating zero date values (like "0000-00-00" or "0000-00-00 00:00:00") as Null values when opening dataset.

mdsoFTStringAsVarchar
When this option is set TMySQLTable.CreateTable method will create VARCHAR columns for **ftString** fields. And if this option is not set TMySQLTable.CreateTable method will create CHAR columns for **ftString** fields.

mdsoCaseInsensitiveLocalSort (since v2.7.6)
When this option is set TMySQLDataset descendants use case-insensitive sorting when using SortBy method or SortFieldNames property. If this option is not set simple binary comparing is performed for strings.

See also: TMySQLTable.CreateTable, TMySQLDataset.SortBy methods, TMySQLDataset.SortFieldNames property

4.1.9. DataSets

Provides an indexed array of all active datasets for a database component.

Syntax:

```
property DataSets[Index: Integer]: TMySQLDataSet;
```

Description:

Use **DataSets** to access active datasets associated with a database component. An active dataset is one that is currently open.

See also: CloseDatasets, Example: DataSetCount,DataSets

4.1.10. DesignOptions

Since v2.7.2

Returns or sets database properties to organize the component behavior at design-time.

Syntax:

```
TMySQLDBDesignOption = (ddoStoreConnected, ddoStorePassword);
TMySQLDBDesignOptions = set of TPSQLDBDesignOption;
property DesignOptions: TMySQLDBDesignOptions;
```

Description:

Set **DesignOptions** to include the desired properties for the TMySQLDatabase behavior at design-time. **DesignOptions** is a set drawn from the following values:

ddoStoreConnected
Enable a saving of the Connected property into a **.dfm** file in design mode.

ddoStorePassword
Save a password into **.dfm** file. It can decrease the database security.

See also: Connected, UserPassword properties

4.1.11. Exclusive

This property is deprecated and left only for compatibility reasons. Changing its value affects nothing. Use MultiThreaded property instead.

See also: MultiThreaded property

4.1.12. Handle

Specifies the database handle.

Syntax:

```
type HDBIDB: Longint;
property Handle: HDBIDB;
```

Description:

Use **Handle** only to bypass **TMySQLDataBase** methods and make direct calls to the directly to the API. Many function calls require a **Handle** parameter. **Handle** is assigned an initial value when a database is opened.

4.1.13. HandleShared

Specifies whether or not to share a database handle.

Syntax:

```
property HandleShared: Boolean;
```

Description:

Use **HandleShared** to indicate that a database component can share its handle. Set **HandleShared** to **True** to avoid namespace conflicts for database components that appear in a remote data module, or that appear in data modules you inherit from the Object Repository.

4.1.14. Host

Sets **Host** on which server is running.

Syntax:

```
property Host: String;
```

Description:

The value of **Host** may be either a hostname or an IP address. If property **Host** is empty string, or the string "localhost", or '127.0.0.1' a connection to the local host is assumed. If the OS supports sockets (Unix) or named pipes (Windows), they are used instead of TCP/IP to connect to the server.

See also: DatabaseName, Port, UserName, UserPassword properties

4.1.15. InTransaction

Indicates whether a database transaction is in progress or not.

Syntax:

```
property InTransaction: Boolean;
```

Description:

Examine **InTransaction** at run-time to determine if a database transaction is currently in progress. **InTransaction** is **True** if a transaction is in progress, **False** otherwise. The value of **InTransaction** cannot be changed directly. Calling StartTransaction sets **InTransaction** to **True**. Calling Commit or Rollback sets **InTransaction** to **False**.

> On contrary to BDE and ODBC you must explicitly specify a TMySQLDataBase component for each MySQL table. Are transactions supported or not in MySQL depends on table type. MyISAM tables (the default type in MySQL) don't support transactions; InnoDB and BDB tables do.

See also: StartTransaction, Commit, Rollback methods

4.1.16. IsSSLUsed

> Since v2.7.1

Read this property value to ensure that SSL encryption is used for connection to MySQL server. This property is read-only.

Syntax:

```
property IsSSLUsed: boolean;
```

Description:

DAC for MySQL supports SSL encryption for connections to MySQL. But there are several conditions for SSL encryption could be used: MySQL server settings, client SSL-libraries availability, proper certificate and key files. **IsSSLUsed** property is used to ensure that SSL encryption is really used after connection is established.

> See SSLCert and SSLKey properties descriptions for details.

See also: SSLCert, SSLKey, ConnectOptions properties

4.1.17. KeepConnection

Specifies whether an application remains connected to a database even if no datasets are open.

Syntax:

```
property KeepConnection: Boolean;
```

Description:

Use **KeepConnection** to specify whether an application remains connected to a database even if no datasets are currently open. When **KeepConnection** is **True** (the default) the connection is maintained. For connections to remote database servers, or for applications that frequently open and close datasets, set **KeepConnection** to **True** to reduce network traffic, speed up applications, and avoid logging in to the server each time the connection is reestablished.

When **KeepConnection** is **False** a connection is dropped when there are no open datasets. Dropping a connection releases system resources allocated to the connection, but if a dataset is later opened that uses the database, the connection must be reestablished and initialized.

4.1.18. LastInsertID

Get last inserted value of AUTO_INCREMENT column from MySQL server.

Syntax:

```
property LastInsertID : Int64;
```

Description:

DAC for MySQL performs 'SELECT LAST_INSERT_ID' query for current database and returns its result as value of this property.

4.1.19. LoginPrompt

Specifies whether a dialog appears immediately before opening a new connection.

Syntax:

```
property LoginPrompt: Boolean;
```

Description:

Set **LoginPrompt** to **True** to provide support when establishing a connection. When **LoginPrompt** is **True**, a dialog appears to prompt users for a name and password. When this dialog appears depends on the type of connection component.

For **TMySQLDatabase**, the dialog appears after the BeforeConnect event and before the AfterConnect event, unless you supply an OnLogin event handler.

If there is an OnLogin event handler, that event occurs in place of the dialog. If correct values for the user name and password are not supplied in the dialog or by the OnLogin event handler, the connection fails.

This OnLogin event does not fire unless **LoginPrompt** is set to **True**.

When **LoginPrompt** is **False**, the application must supply user name and password values programmatically.

> Storing hard-coded user name and password entries as property values or in code for an OnLogin event handler can compromise server security.

> Another way to ask user for username and password is to call ConnectWithConnectionOptionsDialog method.

See also: ConnectWithConnectionOptionsDialog method, OnLogin event

4.1.20. MaxAllowedPacketSize

> Since v2.6.1

Sets *max_allowed_packet* connection parameter value.

Syntax:

```
property MaxAllowedPackedSize : Cardinal; default 16; //in megabytes
```

Description:

Sets *max_allowed_packet* connection parameter value in megabytes. Default value is 16 Mb. Actually this is the maximum size of data in a single field of single row that can be transferred from server to client or from client to server. You can increase it if you are working with large BLOB values. Maximum value is 1024 Mb (for current MySQL versions - 5.0/5.1).

⚠ Please note that this is client-side value. The server has its own *max_allowed_packet* variable, so if you want to handle big packets, you must increase this variable both on the client and on the server.

4.1.21. MultiThreaded

Allows usage of MySQL connection from several threads.

Syntax:

```
property MultiThreaded: Boolean;
```

Description:

If **MultiThreaded** is **True**, the internal connection handle usage is protected with critical sections. This means that you can safely use **TMySQLDataset** descendants (TMySQLTable, TMySQLQuery, TMySQLStoredProc) connected to **TMySQLDatabase** component in different threads of the application.

⚠ Setting **MultiThreaded** property to **True** doesn't mean that several queries will run in several threads simultaneously. They will run one-by-one using the same connection to the database (**TMySQLDatabase** component). If you want to run several queries simultaneously, you'll need separate (**TMySQLDatabase** component) connection for every thread.

4.1.22. Params

Contains database connection parameters for the MySQL server.

Syntax:

```
property Params: TStrings;
```

Description:

Use **Params** to examine or specify database connection parameters, such as server port, server host, user name, and password. **Params** is a list of string items, each representing a different database connection parameter.

💡 At design time double-click a TMySQLDataBase component to invoke the **Database editor** and set **Params**.

4.1.23. Port

Sets server port.

Syntax:

```
property Port: Integer; default 3306
```

Description:

The value will be used as the port number for the TCP/IP connection. If **Port** is set to zero, the default value is used (3306).

See also: DatabaseName, Host, UserName, UserPassword properties

4.1.24. ReadOnly

Specifies that the database connection provides read-only access.

Syntax:

```
property ReadOnly: Boolean;
```

Description:

Use **ReadOnly** to specify whether the database connection should allow the application to update the tables and other metadata in the database. Set **ReadOnly** before opening the database.

When **ReadOnly** is **False** (the default), the application can modify tables and database metadata (like indexes).

When **ReadOnly** is **True**, applications can browse tables but cannot update them. The application is also prevented from creating or deleting metadata objects like tables and indexes.

4.1.25. ServerVersion

Specifies MySQL server version as an integer.

Syntax:

```
property ServerVersion: integer;
```

Description:

Use **ServerVersion** to get server version as an integer.

ServerVersion is represented as **XYYZZ**, where **X** is the major version, **YY** is the release level leftpadded with zero, and **ZZ** is the version number within release series leftpadded with zero. For example, 5.0.22 becomes 50022, 4.1.9 becomes 40109 and so on. Using this property you can just compare servers versions as integers if you need to determine which version is higher.

ServerVersion is equal to **-1** if the **TMySQLDatabase** component is not connected and equal to **0** if the version string can't be successfully parsed.

See also: GetServerInfo method

4.1.26. SSLProperties

Sets options for SSL connection (encrypted protocol).

Syntax:

```
property SSLProperties: TSSLProperties
```

Description:

The value of **ConnectOptions** is usually empty set [], but it can be set to a combination of the following flags in very special circumstances:

SSLCert
Containing the X.509 certificate.

SSLKey
Containing the Key for the X.509 certificate.

SSLCACert
Containing the Key for the authority certificate.

SSLLibName
The SSL library file.

SSLCryptoLibName
The SSL cryptography library file.

SSLCipherList
The cipher suite list.

TLSVersion

TLS protocol version.

To force client library to use SSL encryption, you should add **coSSL** option to ConnectOptions property value . To make sure that SSL encryption is really performed, please check IsSSLUsed property value after connection is established.

To use SSL connections between the MySQL server and client programs, your system must be able to support OpenSSL and your version of MySQL must be 4.0.0 or newer and your MySQL server must be properly configured for SSL support. Please refer to the MySQL manual for details: http://dev.mysql.com/doc/

DAC for MySQL had been tested with OpenSSL 0.9.8, 1.02, and 1.1.1 binaries. Please let us know if you have any problems with newer versions of OpenSSL. Our Support Ticketing system is available at http://microolap.com/support/

Don't forget to add **MySQLTypes** to your uses section.

See also: ConnectOptions, IsSSLUsed properties

4.1.26.1. SSLCert

File containing the X.509 certificate.

Syntax:

```
property SSLCert: String;
```

Description:

SSLCert is a string property that represents the file name that contains the content for the X.509 certificate.

See also: IsSSLUsed, ConnectOptions SSLKey, SSLCACert, SSLLibName, SSLCryptoLibName, SSLCipherList, TLSVersion properties

4.1.26.2. SSLKey

File containing the Key for the X.509 certificate.

Syntax:

```
property SSLKey: String;
```

Description:

SSLKey is a string property that represents the file name containing the contents for the X.509 certificate Key.

See also: IsSSLUsed, ConnectOptions SSLCert, SSLCACert, SSLLibName, SSLCryptoLibName, SSLCipherList, TLSVersion properties

4.1.26.3. SSLCACert

File containing the authority certificate.

Syntax:

```
property SSLCACert: String;
```

Description:

SSLCACert is a string property that represents the file name containing the contents for the authority certificate.

See also: IsSSLUsed, ConnectOptions SSLCert, SSLKey, SSLLibName, SSLCryptoLibName, SSLCipherList, TLSVersion properties

4.1.26.4. SSLLibName

The SSL library file.

Syntax:

```
property SSLLibName: String;
```

Description:

SSLLibName is a string property that represents the file name for the SSL library.
If it is not set, the following default library names will be used for searching: 'libssl-1_1.dll', 'libssl.dll', 'ssleay32.dll'.

See also: IsSSLUsed, ConnectOptions SSLCert, SSLKey, SSLCACert, SSLCryptoLibName, SSLCipherList, TLSVersion properties

4.1.26.5. SSLCryptoLibName

The SSL cryptography library file.

Syntax:

```
property SSLCryptoLibName: String;
```

Description:

SSLLibName is a string property that represents the file name for the SSL cryptography library.
If it is not set, the following default library names will be used for searching: 'libcrypto-1_1.dll', 'libcrypto.dll', 'libeay32.dll'.

See also: IsSSLUsed, ConnectOptions SSLCert, SSLKey, SSLCACert, SSLLibName, SSLCipherList, TLSVersion properties

4.1.26.6. SSLLCipherList

The cipher suite list.

Syntax:

```
property SSLCipherList: String;
```

Description:

SSLCipherList is a string property that represents the cipher suite list.
The cipher list consists of one or more cipher strings separated by colons. Commas or spaces are also acceptable separators, but colons are normally used.
If it is not set a default cipher will be used.

Example of usage:

```
MySQLDatabase1.SSLProperties.SSLCipherList := 'DHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA';
```

See also: IsSSLUsed, ConnectOptions SSLCert, SSLKey, SSLCACert, SSLLibName, SSLCryptoLibName, TLSVersion properties

4.1.26.7. TLSVersion

TLS protocol version.

Syntax:

```
property TLSVersion: TTLSVersion;
type
TTLSVersion = (tlsAuto, tls1, tls1_1, tls1_2);
```

Description:

TLSVersion is an enum property that represents the Transport Layer Security (TLS) cryptographic protocol version that will be used for a session.

tlsAuto
Default and preferred value. Higher possible version is used. For OpenSSL earlier than 1.1 the **tlsAuto = tlsv1_2**. Thus, you will need to specify the concrete version in case of lower TLS version used.

tls1
TLS version 1.0.

tls1_1
TLS version 1.1.

tls1_2
TLS version 1.2.

Don't forget to add **MySQLTypes** to your uses section.

See also: IsSSLUsed, ConnectOptions SSLCert, SSLKey, SSLCACert, SSLLibName, SSLCryptoLibName, TLSCipherList properties

4.1.27. TransIsolation

Specifies the transaction isolation level for transactions.

Syntax:

```
type TTransIsolation = (tiDirtyRead,
                        tiReadCommitted,
                        tiRepeatableRead);
property TransIsolation: TTransIsolation;
```

Description:

Use **TransIsolation** to specify the transaction isolation level for database transactions. Transaction isolation level determines how a transaction interacts with other simultaneous transactions when they work with the same tables, and how much a transaction sees of the work performed by other transactions.

TransIsolation can be any one of the three values summarized in the following table:

Isolation level	Meaning
tiDirtyRead	Permits reading of uncommitted changes made to the database by other simultaneous transactions. Uncommitted changes are not permanent, and might be rolled back (undone) at any time. At this level a transaction is least isolated from the effects of other transactions.
tiReadCommitted	Permits reading of committed (permanent) changes made to the database by other simultaneous transactions. This is the default **TransIsolation** property value.
tiRepeatableRead	Permits a single, one-time reading of the database. The transaction cannot see any subsequent changes made by other simultaneous transactions. This isolation level guarantees that once a transaction reads a record, its view of that record does not change unless it makes a modification to the record itself. At this level, a transaction is most isolated from other transactions.

Applications that use passthrough SQL for handling transactions must pass a transaction isolation level directly to the database server using the appropriate SQL statement.

4.1.28. UserName

The user ID with which you log on to the database.

Syntax:

```
property UserName: String;
```

Description:

You must always supply a user ID when connecting to a database.

See also: DatabaseName, Host, Port, UserPassword properties

4.1.29. UserPassword

To provide a **Password** for the connection.

Syntax:

```
property UserPassword: String;
```

Description:

For security reasons we strongly recommend you to set **Password** for each user.

> ⚠ By default MySQL has user root with empty password and server administering rights. Don't forget to change password for root!

See also: DatabaseName, Host, Port, UserName

4.1.30. Utf8Used

> Since v2.7.0, only for Delphi/C++Builder 2009 and later

Read this property value to ensure that UTF8 character set is used for connection to MySQL server. This property is read-only.

Syntax:

```
property Utf8Used: boolean;
```

Description:

DAC for MySQL supports Unicode strings only if connection character set is UTF8. Utf8Used property is used by DAC for MySQL to enable conversion between UTF8 encoded strings received from server and Delphi's **UnicodeString** data type internal representation.

You can set your connection character set using ConnectionCharacterSet property.

Connection character set can be UTF8 even if you don't set TMySQLDatabase.ConnectionCharacterSet property to 'utf8', e.g. if the server is configured to use UTF8 as default connection character set. You can use Utf8Used property to ensure that UTF8 character set is used in the connection to MySQL server.

> Please read this FAQ section if you want to use Unicode strings in your application: How to use Unicode data in my application?

See also: ConnectionCollation, ConnectionCharacterSet properties

4.1.31. WarningsCount

> Since v2.7.4

Returns number of warnings issued by server for latest query.

Syntax:

```
property WarningsCount : Word;
```

Description:

Examine this property value after running query to check if there are any warnings issued by server for this query. If there are any warnings (**WarningsCount** > 0) then SHOW WARNINGS query could be run to retrieve warnings list.

> Warnings are supported since MySQL 4.1. So if you use MySQL version prior to 4.1 **WarningsCount** property value is always zero.

4.2. Methods

Please see TMySQLDatabase methods short descriptions below:

ApplyUpdates
Reserved for future implementation.

ChangeUser

Changes current user and database without re-establishing database connection. MySQL library *mysql_change_user* function analog.

Close
Closes the connection.

CloseDataSets
Closes all datasets associated with the database component without disconnecting from the database server.

Commit
Permanently stores updates, insertions, and deletions of data associated with the current transaction, and ends the current transactions.

Connect
Connects **TMySQLDatabase** to MySQL database.

ConnectWithConnectionOptionsDialog
Connects **TMySQLDatabase** to MySQL database with "MySQL connection options" dialog.

Create
Creates an instance of a **TMySQLDataBase** component.

Destroy
Destroys the instance of a database component.

Disconnect
Closes connection with database.

Execute
Executes an SQL statement.

GetCharSet
Returns database's code page.

GetClientInfo
Returns a string that represents the client library version.

GetDatabaseCharacterset
Returns character set name string for current database.

GetDatabaseCollation
Returns collation name string for current database.

GetDatabases
Populates a stringlist with the names of persistent MySQL databases.

GetDatabaseSize
Return the current database size in bytes.

GetFieldNames
Populates a string list with the names of fields in a table.

GetFuncNames (deprecated)

Returns list of functions available at the moment in the current database.

GetHostInfo
Returns a string describing the type of connection in use. Deprecated method.

GetIdentifier
Returns quoted identifier if server supports backquote character (`).

GetProtoInfo
Returns the protocol version used by current connection.

GetRoutinesNames
Returns list of routines available at the moment in the current database.

GetServerInfo
Returns a server version number string.

GetServerStat
Returns **MySQLadmin** status info.

GetStoredProcNames (deprecated)
Returns list of stored procedures available at the moment in the current database.

GeTableEngines
Populates a string list with the Engine type of the tables.

GetTableNames
Populates a string list with the names of tables associated with a specified database component.

Kill
Kills a thread.

Open
Opens the connection.

Ping
Pings MySQL server.

Reconnect
Resets the communication channel to the server.

Rollback
Cancels all updates, insertions, and deletions for the current transaction and ends the transaction.

SelectXXX
Methods of this group execute SQL Query and return result as single value casted to XXX type.

Shutdown
Stops MySQL server.

StartTransaction
Begins a new transaction against the database server.

4.2.1. ApplyUpdates

Reserved for future implementation.

4.2.2. ChangeUser

Since v2.7.0

Changes current user and database without re-establishing database connection. MySQL library *mysql_change_user* function analog.

Syntax:

```
function ChangeUser(const aNewUserName : string;
                    const aUserPassword : string;
                    const aDatabaseName : string = '') : boolean;
```

Description:

Use **ChangeUser** method to change current user and database for current database connection.

aNewUserName
Name of user to change with.

aUserPassword
Password of user specified in **aNewUserName** param.

aDatabaseName
Name of database to switch to. If this is empty or omitted then current database remains the same.

ChangeUser method returns **True** if current user and/or database was changed successfully. Method returns **False** without changing current user and/or database if operation fails.

Take a look at MySQL library *mysql_change_user* function description:
http://dev.mysql.com/doc/refman/5.1/en/mysql-change-user.html

4.2.3. Close

Closes the connection.

Syntax:

```
procedure Close;
```

Description:

Call **Close** to disconnect from the source of database information. Before the connection component is deactivated, all associated datasets are closed. Calling **Close** is the same as setting the Connected property to **False**.

In most cases, closing a connection frees system resources allocated to the connection.

> If a previously active connection is closed and then reopened, any associated datasets must be individually reopened; reopening the connection does not automatically reopen associated datasets.

4.2.4. CloseDataSets

Closes all datasets associated with the database component without disconnecting from the database server.

Syntax:

```
procedure CloseDatasets;
```

Description:

Call **CloseDataSets** to close all active datasets without disconnecting from the database server. Ordinarily, when an application calls Close, all datasets are closed, and the connection to the database server is dropped.

Calling **CloseDataSets** instead of Close ensures that an application can close all active datasets without having to reconnect to the database server at a later time.

4.2.5. Commit

Permanently stores updates, insertions, and deletions of data associated with the current transaction, and ends the current transactions.

Syntax:

```
procedure Commit;
```

Description:

Call **Commit** to permanently store to the database server all updates, insertions, and deletions of data associated with the current transaction and then end the transaction. The current transaction is the last transaction started by calling **StartTransaction**.

> Before calling **Commit**, an application may check the status of the InTransaction property. If an application calls **Commit** and there is no current transaction, an exception is raised.

See also: StartTransaction, Rollback methods, InTransaction property

4.2.6. Connect

> Since v2.5.3

Connects **TMySQLDatabase** to MySQL database.

Syntax:

```
procedure Connect;
```

Description:

Call **Connect** to set the **Connected** property for the component to **True**. When **Connected** is **True**, component is connected to database and ready to exchange data with it.

See also: Connected property and Disconnect method

4.2.7. ConnectWithConnectionOptionsDialog

> Since v2.5.3

Connects **TMySQLDatabase** to MySQL database with "MySQL connection options" dialog.

Syntax:

```
function ConnectWithConnectionOptionsDialog : boolean;
```

Description:

Method returns **True** if connection was established successfully and **False** otherwise.

Call **ConnectWithConnectionOptionsDialog** to set the Connected property for the component to **True**. This method is similar to Connect method but it shows "MySQL connection options" dialog before connect to database. This allows to change connection parameters, for example to set username or password.

See also: Connected property, Connect method, OnLogin event

4.2.8. Create

Creates an instance of a TMySQLDataBase component.

Syntax:

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

Description:

Call **Create** to instantiate a database component at runtime. An application can create a database component in order to control the component's existence and set its properties and events, or an application can let Delphi create temporary database components as needed at runtime.

Create instantiates a database component and:

- Adds this component to the list of database components.
- Creates an empty list of dataset components for the. DataSets property.
- Creates an empty string list for the Params property.
- Sets the LoginPrompt property and the KeepConnection property to **True**.
- Sets the TransIsolation property to **tiReadCommitted**.

4.2.9. Destroy

Destroys the instance of a database component.

Syntax:

```
destructor Destroy;
```

Description:

Do not call **Destroy** directly in an application. Instead, call **Free**, which verifies that the database reference is not **nil** before calling **Destroy**.

Destroy closes all active datasets and disconnects from the database server, if necessary. It then frees the string resources allocated for the Params and DataSets properties before calling its inherited destructor.

4.2.10. Disconnect

Since v2.5.3

Closes connection with database.

Syntax:

```
procedure Disconnect;
```

Description:

Call **Disconnect** to set the Connected property of a component to **False**. When Connected is **False**, the connection to database is closed.

See also: Connected property and Connect method.

4.2.11. Execute

Executes an SQL statement.

Syntax:

```
function Execute(const SQL: String;
                       Params: TParams  = nil;
                        Cache: Boolean  = False): Integer;
```

Description:

Use **Execute** to execute an SQL statement against the database without the overhead of using a TMySQLQuery object.

SQL
A **String** value containing the statement to be executed.

Params

A value of type **TParams** and lists any parameters used by the SQL statement. Parameter binding is by index only (not by name), so the order of parameters is important and the order of the **TParam** objects in **Params** corresponds to the order of the parameters in the SQL statement. Use properties and methods of **TParams** to create a **TParams** object and add one **TParam** object for each parameter. Use properties and methods of **TParam** like the **AsString** property to give each parameter a value prior to calling Execute. If the SQL statement does not include any parameters, pass a **nil** value for **Params**.

Cache

Specifies whether the prepared SQL statement is cached for reuse within the current transaction. Caching statements can speed their processing if they are used more than once in a transaction.

Execute returns the number of records affected by executing the SQL statement.

Execute doesn't open or stores any resultsets returned by query. All resultsets are ignored.

See also: SelectXXX methods group

4.2.12. GetCharSet

Returns database code page.

Syntax:

```
function GetCharSet: TConvertChar;
type
TConvertChar  =  (ccUndefine,
                  cc8859_1,
                  cc8859_10,
                  cc8859_13,
                  cc8859_14,
                  cc8859_15,
                  cc8859_2,
                  cc8859_3,
                  cc8859_4,
                  cc8859_5,
                  cc8859_6,
                  cc8859_7,
                  cc8859_8,
                  cc8859_9,
                  ccCp1250,
                  ccCp1251,
                  ccCp1252,
                  ccCp1253,
                  ccCp1254,
                  ccCp1255,
                  ccCp1256,
                  ccCp1257,
                  ccCp1258,
                  ccCp424,
```

```
                ccCp437,
                ccCp500,
                ccCp737,
                ccCp775,
                ccCp850,
                ccCp852,
                ccCp855,
                ccCp856,
                ccCp857,
                ccCp860,
                ccCp861,
                ccCp862,
                ccCp863,
                ccCp864,
                ccCp865,
                ccCp866,
                ccCp869,
                ccCp874,
                ccCp875,
                ccKoi8_r,
                ccUtf8);
```

4.2.13. GetClientInfo

Returns a string that represents the client library version.

Syntax:

```
function GetClientInfo: String;
```

See also: GetProtoInfo, GetServerInfo, GetServerStat methods, ServerVersion property.

4.2.14. GetDatabaseCharacterset

Since v2.6.0

Returns character set name string for current database.

Syntax:

```
function GetDatabaseCharacterset: String;
```

Description:

GetDatabaseCharacterset method returns character set name as it have been reported by server. For example: cp1251.

This method returns *'character_set'* server variable's value for MySQL earlier than 4.1.1 and *character_set_database* server variable value for MySQL greater than 4.1.1

Please read this FAQ section if you want to use Unicode strings in your application: How to use Unicode data in my application?

See also: GetDatabaseCollation method

4.2.15. GetDatabaseCollation

Since v2.6.0

Returns collation name string for current database.

Syntax:

```
function GetDatabaseCollation: String;
```

Description:

GetDatabaseCollation method returns collation name as it have been reported by server. For example: *cp1251_general_cs*.

This method returns *'collation_database'* server variable value for MySQL greater than 4.1.1 and empty string value for MySQL earlier 4.1.1

See also: GetDatabaseCharacterset method

4.2.16. GetDatabases

Populates a stringlist with the names of persistent MySQL databases.

Syntax:

```
procedure GetDabases(Pattern: String; List : TStrings);
```

4.2.17. GetDatabaseSize

Return the current database size in bytes.

Syntax:

```
function GetDatabaseSize : Int64;
```

Description:

Use these method to get the size current database in bytes. If there is no connection to the database, the function returns **0**.

Exampls:

```
ShowMessage('Size current database is ' +
IntToStr(MySQLDatabase1.GetDatabaseSize)
                + ' bytes');
```

4.2.18. GetFieldNames

Populates a string list with the names of fields in a table.

Syntax:

```
procedure GetFieldNames(const TableName: String;
                                    List: TStrings);
```

Description:

Call **GetFieldNames** to retrieve a list of fields in a table.

TableName
Names the table whose field names you want added to the list.

List
A **TStrings** descendant that receives the field names. Any existing strings are deleted from the list before **GetFieldNames** adds the names of all the fields in **TableName**.

Example:

The following line fills a list box with the names of all fields in the table:

```
Database1.GetFieldNames('Employee', ListBox1.Items);
```

4.2.19. GetFuncNames

Returns list of functions available at the moment in the current database.

This procedure is deprecated. Please use GetRoutinesNames instead.

Syntax:

```
GetFuncNames(Pattern: string; List: TStrings);
```

Parameters:

Pattern
Use **Pattern** parameter to set wildcard matching for functions names.

List
A string list object, created and maintained by the application, in which the functions names will be return to.

Description:

Call **GetFuncNames** to obtain available functions list. Strings are sorted by names.

Example:

```
var str:string; list:TStringList; i:integer;
 begin
  MySQLDatabase1.GetFuncNames('', list);
  str := 'Existing functions:'+#13#10;
  for i := 0 to list.Count-1 do
       str := str + '    ' + list[i] + #13#10;
  ShowMessage(str);
 end;
```

4.2.20. GetHostInfo

Deprecated method

Returns a string describing the type of connection in use, including the server host name.

Syntax:

```
function GetHostInfo: String;
```

Description:

Returns an empty string for latest DAC for MySQL versions.

See also: GetProtoInfo, GetServerInfo, GetServerStat, GetClientInfo methods, ServerVersion property.

4.2.21. GetIdentifier

Since v2.6.0

Returns quoted identifier if server supports backquote character (`).

Syntax:

```
function GetIdentifier(aIdentName : string) : string;
```

Description:

GetIdentifier method returns value of **aIdentName** parameter quoted with backquote characters (`) if server version is equal or greater 3.23.6. This method is useful if you want to ensure, that your identifier is acceptable for server.

```
Example:
var s, r : string;
...
s := 'MyTableName';
r := mySQLDatabase1.GetIdentifier(s);
```

Value of r variable will remain *MyTableName* for MySQL before 3.23.6 and will be *`MyTableName`* otherwise.

4.2.22. GetProtoInfo

Returns the protocol version used by current connection.

Syntax:

```
function GetProtoInfo: Cardinal;
```

See also: GetServerInfo, GetServerStat, GetClientInfo methods, ServerVersion property.

4.2.23. GetRoutinesNames

Returns list of routines available at the moment in the current database.

Syntax:

```
GetRoutinesNames(Pattern: string; List: TStrings; SrtType:
TMySQLSelectRoutinesType = srtAll);
TMuSQLSelectRoutinesType = (srtProc, srtFunc, srtAll);
```

Parameters:

Pattern
Use **Pattern** parameter to set wildcard matching for functions names.

List
A string list object, created and maintained by the application, in which the functions names will be return to.

SrtType
Specifies which type of routines will be returns.

Description:

Call to **GetRoutinesNames** to obtain available routines list. Strings are sorted by names.

Example:

```
var str:string; list:TStringList; i:integer;
 begin
  MySQLDatabase1.GetRoutinesNames('', list, TMySQLSelectRoutinesType.srtFunc);
  str := 'Existing functions:'+#13#10;
  for i := 0 to list.Count-1 do
       str := str + '    ' + list[i] + #13#10;
  ShowMessage(str);
 end;
```

4.2.24. GetServerInfo

Returns the server version string.

Syntax:

```
function GetServerInfo: String;
```

Description:

GetServerInfo method returns version number string as it have been reported by server. For example: 5.1.11-beta.

See also: GetProtoInfo, GetServerStat, GetClientInfo methods, ServerVersion property.

4.2.25. GetServerStat

Returns **MySQLadmin** status info.

Syntax:

```
function GetServerStat: String;
```

Description:

Returns a character string containing information similar to that provided by the **MySQLadmin** status command. This string includes uptime in seconds and the numbers of:

- Running threads;
- Questions;
- Reloads;
- Open tables.

See also: GetProtoInfo, GetServerInfo, GetClientInfo methods, ServerVersion property.

4.2.26. GetStoredProcNames

Returns list of stored procedures available at the moment in the current database.

This procedure is deprecated. Please use GetRoutinesNames instead.

Syntax:

```
GetStoredProcNames(Pattern: string; List: TStrings);
```

Parameters:

Pattern
Use **Pattern** parameter to set wildcard matching for procedure names.

List
A string list object, created and maintained by the application, in which the stored procedures names will be return to.

Description:

Call **GetStoredProcNames** to obtain available stored procedures list. Strings are sorted by names.

Examples:

```
var str:string; list:TStringList; i:integer;
 begin
  MySQLDatabase1.GetStoredProcNames('', list);
  str := 'Existing procedures:'+#13#10;
  for i := 0 to list.Count-1 do
       str := str + '    ' + list[i] + #13#10;
  ShowMessage(str);
 end;
```

4.2.27. GetTableEngines

Populates a string list with the **Engine type** of the tables.

Syntax:

```
procedure GetTableEngines(TableList: TStrings; TableEnginesList: TStrings);
```

Parameters:

TableList
A string list object with tables for which you need to know **Engine type**.

TableEnginesList
A string list object in which the **Engine type** of the tables will be return.

Description:

Call to **GetTableEngines** to obtain tables **Engine type**. **TableEnginesList must be previously created.**

4.2.28. GetTableNames

Populates a string list with the names of tables associated with a specified database component.

Syntax:

```
procedure GetTableNames(Pattern: String; List: TStrings);
```

Description:

Call **GetTableNames** to retrieve a list of the tables associated with a given database.

Pattern specifies a delimiter string that restricts the tables returned to those that match the string. **Pattern** can include wildcard symbols. Pass an empty **Pattern** string to match all tables not restricted by other criteria.

List is a string list object, created and maintained by the application, into which to return the table names.

4.2.29. Kill

Kills a thread.

Syntax:

```
procedure Kill(PID : Integer);
```

Description:

Asks the server to kill the thread specified by PID.

4.2.30. Open

Opens the connection.

Syntax:

```
procedure Open;
```

Description:

Call **Open** to establish a connection to the source of database information. **Open** sets the Connected property to **True**.

4.2.31. Ping

Pings MySQL server.

Syntax:

```
function Ping : Integer;
```

Description:

Checks if the connection to the server is working. If it has gone down, an automatic reconnection is attempted. Returns **1** if the server is alive and **0** if an error occurred.

Negative result (0) does not indicate whether the MySQL server itself is down; the connection might be broken for other reasons such as network problems.

This function can be used by clients that remain idle for a long time to check if the server has closed the connection and reconnect if necessary.

4.2.32. Reconnect

Resets the communication channel to the server.

Syntax:

```
procedure Recconnect;
```

Description:

This function will close the connection to the server and attempt to reestablish a new connection to the same server, using all the same parameters previously used. This may be useful for error recovery if a working connection is lost.

When calls **Reconnect** generated event **OnReconnect**.

Before calling **Reconnect**, **TMySQLDatabase** must be Connected to server. This means that **TPSQLDatabase.Open** method was called before, or property **Connected** was set to **True**. If an application calls **Reset** and there was no active connection, an exception is raised.

See also: TMySQLDatabase.Events.OnReconnect

4.2.33. Rollback

Cancels all updates, insertions, and deletions for the current transaction and ends the transaction.

Syntax:

```
procedure Rollback;
```

Description:

Call **Rollback** to cancel all updates, insertions, and deletions for the current transaction and to end the transaction. The current transaction is the last transaction started by calling StartTransaction.

> Before calling **Rollback**, an application may check the status of the InTransaction property. If an application calls Rollback and there is no current transaction, an exception is raised.

See also: Commit, StartTransaction methods InTransaction property

4.2.34. SelectXxx

> Since v2.5.5, v2.6.1 (SelectInt64 and SelectDateTime)

Methods of this group execute SQL query and return result as single value casted to XXX type.

Syntax:

```
function SelectString(aSQL : string; var IsOk : boolean; aFieldNumber : integer =
0):string;
function SelectString(aSQL : string; var IsOk : boolean; aFieldName :
string):string;overload;
function SelectStringDef(aSQL : string; aDefaultValue : string; aFieldNumber :
integer = 0):string;
function SelectStringDef(aSQL : string; aDefaultValue : string; aFieldName :
string):string;
function SelectInteger(aSQL : string; var IsOk : boolean; aFieldNumber : integer
= 0):integer;
function SelectInteger(aSQL : string; var IsOk : boolean; aFieldName :
string):integer;
function SelectIntegerDef(aSQL : string; aDefaultValue : integer; aFieldNumber :
integer = 0):integer;
function SelectIntegerDef(aSQL : string; aDefaultValue : integer; aFieldName :
string):integer;
function SelectInt64(aSQL : string; var IsOk : boolean; aFieldNumber : integer =
0):int64;
function SelectInt64(aSQL : string; var IsOk : boolean; aFieldName :
string):int64;
function SelectInt64Def(aSQL : string; aDefaultValue : int64; aFieldNumber :
integer = 0):int64;
function SelectInt64Def(aSQL : string; aDefaultValue : int64; aFieldName :
string):int64;
function SelectDouble(aSQL : string; var IsOk : boolean; aFieldNumber : integer =
0):double;
function SelectDouble(aSQL : string; var IsOk : boolean; aFieldName :
string):double;
function SelectDoubleDef(aSQL : string; aDefaultValue : double; aFieldNumber :
integer = 0):double;
```

```
function SelectDoubleDef(aSQL : string; aDefaultValue : double; aFieldName :
string):double;
function SelectDateTime(aSQL : string; var IsOk : boolean; aFieldNumber : integer
= 0):TDateTime;
function SelectDateTime(aSQL : string; var IsOk : boolean; aFieldName :
string):TDateTime;
function SelectDateTimeDef(aSQL : string; aDefaultValue : TDateTime; aFieldNumber
: integer = 0):TDateTime;
function SelectDateTimeDef(aSQL : string; aDefaultValue : TDateTime; aFieldName :
string):TDateTime;
```

Description:

Use these methods to execute an SQL statement against the database without the overhead of using a TMySQLQuery object. Methods return single value as a value of the field specified by number (**aFieldNumber** parameter) or by name (**aFieldName** parameter) in the first row of the first resultset. If the **aFieldNumber** parameter is omitted, methods return the value of the first field.

If the query doesn't return at least one row or specified field is not found, the **IsOk** param is set to **False** (**SelectXxx** methods) or **aDefaultValue** param value is returned (**SelectXxxDef** methods).

Parameters:

aSQL
A String value containing the statement to be executed.

IsOk
(**SelectXxx** methods) variable is set to **True** after successful execution, or **False** if requested value couldn't be fetched.

aDefaultValue
(**SelectXxxDef** methods) value to return by method if requested value can't be fetched.

aFieldNumber
Field number to return its value by method. Fields are numbered from zero. If this parameter is omitted, then it is assumed to be **0**. In this case the value of the first field will be returned.

aFieldName
Field name to return its value by method.

Examples:

This example will show MySQL server version (like TMySQLDatabase.GetServerInfo method):

```
ShowMessage(mySQLDatabase1.SelectStringDef('SELECT version', 'Something wrong
happend!'));
```

This example will show current UNIX timestamp value returned by MySQL server:

```
ShowMessage(IntToStr(mySQLDatabase1.SelectIntDef('SELECT unix_timestamp', -1)));
```

See also: Execute method

4.2.35. Shutdown

Stops MySQL server.

Syntax:

```
function Shutdown: integer;
```

Description:

If the function succeeds, the returned value is "**0**", or any other value on error.

Connected user must have SHUTDOWN rights.

4.2.36. StartTransaction

Begins a new transaction against the database server.

Syntax:

```
procedure StartTransaction;
```

Description:

Call **StartTransaction** to begin a new transaction against the database server. Before calling **StartTransaction**, an application should check the status of the InTransaction property and adjust the setting of the TransIsolation property as desired.

If InTransaction is **True**, indicating that a transaction is already in progress, a subsequent call to **StartTransaction** without first calling Commit or Rollback to end the current transaction raises an exception.

Updates, insertions, and deletions that take place after a call to **StartTransaction** are held by the server until an application calls Commit to save the changes or Rollback is to cancel them.

See also: Commit, Rollback methods InTransaction property

4.3. Events

Please see TMySQLDatabase events short descriptions below:

AfterConnect
Occurs after a connection is established.

AfterDisconnect
Occurs after the connection closes.

BeforeConnect
Occurs immediately before establishing a connection.

BeforeDisconnect
Occurs immediately before the connection closes.

OnConnectionFailure
Occurs on a database connection error.

OnLogin
Occurs when an application connects to a database.

OnReconnect
Occurs after a connection is silently re-established after connection failure.

4.3.1. AfterConnect

Occurs after a connection is established.

Syntax:

```
property AfterConnect: TNotifyEvent;
```

Description:

Write an **AfterConnect** event handler to take application-specific actions immediately after the connection component opens a connection to the source of database information.

4.3.2. AfterDisconnect

Occurs after the connection closes.

Syntax:

```
property AfterDisconnect: TNotifyEvent;
```

Description:

Write an **AfterDisconnect** event handler to take application-specific actions after the connection component drops a connection.

4.3.3. BeforeConnect

Occurs immediately before establishing a connection.

Syntax:

```
property BeforeConnect: TNotifyEvent;
```

Description:

Write a **BeforeConnect** event handler to take application-specific actions before the connection component opens a connection to the source of database information.

4.3.4. BeforeDisconnect

Occurs immediately before the connection closes.

Syntax:

```
property AfterDisconnect: TNotifyEvent;
```

Description:

Write a **BeforeDisconnect** event handler to take application-specific actions before dropping a connection.

4.3.5. OnConnectionFailure

Occurs on a database connection error.

Syntax:

```
type TConnectionFailureEvent = procedure(Connection : TMySQLDatabase;
                                         Error : String) of Object;
property OnConnectionFailure : TConnectionFailureEvent;
```

Description:

Write an **ConnectionFailure** event handler to take specific actions when a database connection error occurs.

4.3.6. OnLogin

Occurs when an application connects to a database.

Syntax:

```
type
   TMySQLDataBaseLoginEvent = procedure(Database: TMySQLDataBase;
                                        LoginParams: TStrings)of object;
property OnLogin: TMySQLDataBaseLoginEvent;
```

Description:

Write an **OnLogin** event handler to take specific actions when an application attempts to connect to a database. By default, a database login is required. The current USERNAME is read from the Params property, and a standard Login dialog box opens. The dialog prompts for a user name and password combination, and then uses the values entered by the user to set the UID and PWD values in the Params property. These values are then passed to the remote server.

Applications that provide alternative **OnLogin** event handlers must set the UID and PWD values in **LoginParams**. **LoginParams** is a temporary string list and is freed automatically when no longer needed.

This event handler is called only if LoginPrompt property value is **True**.

Example:

```
procedure TForm1.mySQLDatabase1Login(Database: TMySQLDatabase; LoginParams:
TStrings);
var
  vUserName, vPassword : string;
begin
  vUserName := 'root';           //most probably you want to ask user for user
name and password
  vPassword := 'strongpassword'; //with some GUI instead of just assigning it
here
  LoginParams.Add('UID=' + vUserName);
  LoginParams.Add('PWD=' + vPassword);
end;
```

Another way to ask user for username and password is to call ConnectWithConnectionOptionsDialog method

See also: LoginPrompt property, ConnectWithConnectionOptionsDialog method

4.3.7. OnReconnect

Since v2.6.1

Occurs after a connection is silently re-established after connection failure.

Syntax:

```
TReconnectEvent = procedure(Connection : TMySQLDatabase) of Object;
property OnReconnect: TReconnectEvent;
```

Description:

Write an **OnReconnect** event handler to take application-specific actions when a connection to database server was dropped and restored for some reasons (for example due to network fail). If TMySQLDatabase components detects connection failure it tries to re-establish it silently. Usually you don't have to worry about this. But this event may be very useful if your application depends on temporary tables which will be dropped after such re-connect.

5. TMySQLDataset

TMySQLDataset encapsulates database connectivity for descendant dataset objects.

TMySQLDataSet defines database-related connectivity properties and methods for a dataset.

Applications never use **TMySQLDataset** objects directly. Instead they use the descendants of **TMySQLDataSet**, such as TMySQLQuery, TMySQLTable and TMySQLStoredProc, which inherit its database-related properties and methods.

TMySQLDataset descendants provide BDE-like functionality and fully compatible with **TDatasource** and visual DB-controls. If you need to fetch some data without displaying them you can use high-performance TMySQLDirectQuery component.

See also: Properties, Methods, Events

5.1. Properties

Please see TMySQLDataset properties short descriptions below:

Derived from TDataSet:

Active
Specifies whether or not a dataset is open.

AutoCalcFields
Determines when the **OnCalcFields** event is triggered.

Bof
Indicates whether or not a cursor is positioned at the first record in a dataset.

Bookmark
Specifies the current bookmark in the dataset.

CachedUpdates
Does not affect on dataset behavior.

CanModify
Indicates whether the database underlying a dataset permits write access to data..

DefaultFields
Indicates whether a dataset's underlying field components are generated dynamically when the dataset is opened.

Eof
Indicates whether or not a cursor is positioned at the last record in a dataset.

FieldCount
Indicates the number of field components associated with the dataset.

FieldDefList
Points to the list of field definitions for the dataset.

FieldList
Lists the field components of a dataset.

Fields
Lists all non-aggregate field components of the dataset.

FieldValues
Provides access to the values for all fields in the active record for the dataset.

Found
Indicates whether or not moving to a different record is successful.

Modified
Indicates whether the active record is modified.

Name
Designates the name of the dataset as referenced by other components.

ObjectView
Specifies whether fields are to be stored hierarchically or flattened out in the **Fields** property.

SparseArrays
Determines whether a unique **TField** object is created for each element of an array field.

State
Indicates the current operating mode of the dataset.

In TMySQLDataSet:

AllowSequenced
Determines that database records can be located by sequence numbers.

AutoRefresh
Specifies whether server-generated field values are refetched automatically.

AvailableResultsetCount
Indicates count of resultsets available to fetch from multiresultset query or stored procedure. This property is useful when query or stored procedure returns more than one dataset.

BlockReadSize
Determines how many record buffers are read in each block.

CacheBlobs
Determines whether BLOB fields are cached in memory.

Database
Specifies the database component for which this dataset represents one or more tables.

Filter
Specifies the text of the current filter for a dataset.

Filtered
Specifies whether filtering is active for a dataset.

FilterOptions
Specifies whether filtering is case insensitive, and whether or not partial comparisons are permitted when filtering records.

KeySize
Specifies the size of the key for the current index of the dataset.

LastInsertID
Get last inserted value of AUTO_INCREMENT column from MySQL server

MultiResultsetNo
Specifies the resultset to associate with component when it become active.

RecNo
Indicates the current record in the dataset.

RecordCount
Indicates the total number of records associated with the dataset.

RecordSize
Indicates the size of a record in the dataset.

RefreshDelete
Specifies whether a dataset should delete a record locally when a database record is not found at refreshing.

SortFieldNames
Specifies field names and sorting order to sort opened dataset by these fields on the client side without refetching data from server.

StatementID
Returns the statement id of the prepared query.

UpdateMode
Determines how MySQL finds records when updating to an SQL database.

UpdateObject
Specifies the update object component used to update a read-only result set.

5.1.1. Active

Specifies whether or not a dataset is open.

Syntax:

```
property Active: Boolean;
```

Description:

Use **Active** to determine or set a dataset's connection to data in a database. When **Active** is **False**, the dataset is closed; the dataset cannot read data from or write data to the database. When **Active** is **True**, data can be read from and written to the database.

Setting Active to **True**:

- Triggers the **BeforeOpen** event handler if one is defined for the dataset.
- Sets the dataset state to **dsBrowse**.
- Opens a cursor into the dataset.

- Triggers the **AfterOpen** event handler if one is defined for the dataset.
- If an error occurs while opening the dataset, dataset state is set to **dsInactive**, and the cursor is closed.

An application must set **Active** to **False** before changing other properties that affect the status of the database or the controls that display data in an application.

> Calling the **Open** method sets **Active** to **True**; calling the **Close** method sets **Active** to **False**.

5.1.2. AllowSequenced

Determines that database records can be located by sequence numbers.

Syntax:

```
property AllowSequenced: Boolean;
```

Description:

AllowSequenced determines that database records can be located by sequence numbers. When **AllowSequenced** is **True**, the dataset can navigate directly to a specific record by setting the **RecNo** property.

If **AllowSequenced** is **False**, the only way to navigate to a specific record is to start at the beginning and count records. **AllowSequenced** indicates whether sequence numbers are available for database records.

5.1.3. AutoCalcFields

Determines when the **OnCalcFields** event is triggered.

Syntax:

```
property AutoCalcFields: Boolean;
```

Description:

Set **AutoCalcFields** to control when the **OnCalcFields** event is triggered to update calculated fields during dataset processing. A calculated field is one that derives its value from the values of one or more fields in the active record, sometimes with additional processing.

When **AutoCalcFields** is **True** (the default), **OnCalcFields** is triggered when:

- The dataset is opened.
- The dataset is put into **dsEdit** state.
- Focus moves from one visual control to another, or from one column to another in a data-aware grid and modifications were made to the record.
- A record is retrieved from a database.

If an application permits users to change data, **OnCalcFields** is frequently triggered. In these cases an application may set **AutoCalcFields** to **False** to reduce the frequency with which **AutoCalcFields** is called. When **AutoCalcFields** is **False**, **OnCalcFields** is not called when changes are made to individual fields within a record.

5.1.4. AutoRefresh

Specifies whether server-generated field values are refetched automatically.

Syntax:

```
property AutoRefresh: Boolean;
```

Description:

When **AutoRefresh** is **False** (the default), values that the server creates for autoincrement fields and fields with default values when a record is posted are not automatically refetched by the dataset. Instead, the application must call the dataset's **Refresh** method to update these field values. When **AutoRefresh** is **True**, these field values are automatically refreshed without an explicit call to the **Refresh** method.

5.1.5. AvailableResultsetCount

Since v2.5.3

Indicates count of resultsets available to fetch from multiresultset query or stored procedure. This property is useful when query or stored procedure returns more than one dataset.

Syntax:

```
property AvailableResultsetCount : integer
```

Description:

Inspect **AvailableResultsetCount** property to check number of available resultsets returned from multiresultset query or stored procedure. This property is **0** for closed dataset and it is always **1** for opened **TMySQLTable** components. You can use MultiResultsetNo property to fetch other resultset than first for TMySQLQuery and TMySQLStoredProc components.

See also: MultiResultsetNo property

5.1.6. BlockReadSize

Determines how many record buffers are read in each block.

Syntax:

```
property BlockReadSize: Integer;
```

Description:

Set **BlockReadSize** when you need to scan through the entire dataset quickly. When **BlockReadSize** is greater than zero, and Next is called, data-aware controls are not updated, and data events are not triggered. Set **BlockReadSize** to zero to disable block read mode. The dataset State is **dsBlockRead** when **BlockReadSize** is greater than zero.

5.1.7. Bof

Indicates whether or not a cursor is positioned at the first record in a dataset.

Syntax:

```
property Bof: Boolean;
```

Description:

Test **Bof** (beginning of file) to determine if the cursor is positioned at the first record in a dataset. If **Bof** is **True**, the cursor is unequivocally on the first row in the dataset.

Bof is **True** when an application:

- Opens a dataset.

- Calls a dataset's **First** method.
- Call a dataset's **Prior** method, and the method fails (because the cursor is already on the first row in the dataset).
- Calls **SetRange** on an empty range or dataset.

Bof is **False** in all other cases.

See also: **Example: GetBookmark, GotoBookmark,FreeBookmark, FindPrior, Value, OnDataChange,BOF**

5.1.8. Bookmark

Specifies the current bookmark in the dataset.

Syntax:

```
type TBookmarkStr: String;
property Bookmark: TBookmarkStr;
```

Description:

Bookmark gets or sets the current bookmark in a dataset. A bookmark marks a location in a dataset so that an application can easily return to that location quickly.

An application can read **Bookmark** to retrieve the bookmark associated with the current record, and it can position the cursor in the dataset by assigning a saved bookmark value to this property.

5.1.9. CacheBlobs

Determines whether BLOB fields are cached in memory.

Syntax:

```
property CacheBlobs: Boolean;
```

Description:

Use **CacheBlobs** to specify whether or not to store BLOB images in memory to improve performance when scrolling through records that display BLOB images. **CacheBlobs** is **True** by default, meaning that BLOBs are cached in memory. If an application does not need to display the BLOBs associated with records, then set **CacheBlobs** to **False** to conserve system resources.

5.1.10. CachedUpdates

Does not affect on dataset behavior!

Syntax:

```
property CachedUpdates: Boolean;
```

Description:

DAC for MySQL does not support cached updates at all. This property was leaved only for compatibility with BDE classes. Setting it to **True** does not affect dataset behavior at all.

5.1.11. CanModify

Indicates whether the database underlying a dataset permits write access to data.

Syntax:

```
property CanModify: Boolean;
```

Description:

Examine **CanModify** to determine if the database underlying a dataset permits write access to data. When a database connection is established, a dataset typically requests write access. **CanModify** indicates whether or not write access is granted. If **CanModiy** is **True**, data can be modified and written to the database server. If **CanModify** is **False**, data can be viewed, but not modified.

You can set RequestLive property of TMySQLQuery component to **True** to try to get "live" (updatable) resultset from SQL query.

See also: TMySQLQuery.RequestLive property

5.1.12. Database

Specifies the **TMySQLDatabase** component this component connects to to perform database operations.

Syntax:

```
property Database: TMySQLDatabase;
```

Description:

Use Database property to access the connection and some other properties, events, and methods of the database component associated with this dataset.

In design-time you may choose database from drop-down list for given **MySQLTable** or **MySQLQuery**.

> Since v2.7.1 DAC for MySQL tries to assign this property value to last added TMySQLDatabase component in design-time.

Example:

```
// Do a transaction
with MySQLTable1.Database do
begin
  StartTransAction;
  // transactions supporting depends on table's type.
  // MyISAM tables (default type in MySQL) don't
  // support transactions, InnoDB, Falcon and BDB tables do.
  // Post some records with MySQLTable1
  Commit;
end;
```

See also: TMySQLDatabase component

5.1.13. DefaultFields

Indicates whether a dataset's underlying field components are generated dynamically when the dataset is opened.

Syntax:

```
property DefaultFields Boolean;
```

Description:

Read **DefaultFields** to determine whether a dataset uses dynamically generated field components or persistent field components. If **DefaultFields** is **True**, the dataset uses dynamically allocated field components. If **DefaultFields** is **False**, the dataset uses persistent field components.

Unless persistent field components are assigned to a dataset at design time using the Fields editor, the dataset creates dynamic field components based on the structure of its underlying database table or tables when it is opened.

5.1.14. Eof

Indicates whether or not a cursor is positioned at the last record in a dataset.

Syntax:

```
property Eof: Boolean;
```

Description:

Test **Eof** (end-of-file) to determine if the cursor is positioned at the last record in a dataset. If **Eof** is **True**, the cursor is unequivocally on the last row in the dataset.

Eof is **True** when an application:

- Opens an empty dataset.
- Calls a dataset **Last** method.
- Call a dataset **Next** method, and the method fails (because the cursor is already on the last row in the dataset).
- Calls **SetRange** on an empty range or dataset.

Eof is **False** in all other cases.


If both **Eof** and **Bof** are **True**, the dataset or range is empty.

See also: Example: DisableControls, EnableControls, Eof

5.1.15. FetchOnDemand

Specifies whether the rows transferred across the network by portions.

Syntax:

```
property FetchOnDemand : boolean default false;
```

Description:

Use **FetchOnDemand** property to activate the rows transfer across the network by portions specified by **FetchRows** property.

When setting **FetchOnDemand** = **True** please keep in mind that DAC for MySQL will create additional session in order not to block current session. But this can cause the following problems:

- Each additional session runs outside of the transaction context, thus TmySQLDatabase.Commit and TmySQLDatabase.Rollback operations in main session won't apply changes made in additional sessions.
- Temporary tables created in one session are not accessible from other sessions, therefore simultaneous using of **FetchOnDemand = True** and temporary tables is impossible.

See also: FetchRows property

5.1.16. FetchRows

Specifies the number of rows to be transferred across the network at the same time.

Syntax:

```
property FetchRows : integer default 25;
```

Description:

Use **FetchRows** property for specifying the number of rows to be transferred across the network at the same time when **FetchOnDemand** = **True**.

See also: FetchOnDemand property

5.1.17. FieldCount

Indicates the number of field components associated with the dataset.

Syntax:

```
property FieldCount: Integer;
```

Description:

Examine **FieldCount** to determine the number of fields listed by the Fields property. For datasets with dynamically created fields, **FieldCount** may differ each time a dataset is opened. For datasets with persistent fields, **FieldCount** should be unchanged each time a dataset is open.

FieldCount includes only the fields listed by the **Fields** property.

See also: Example: FieldCount, Fields, FieldName

5.1.18. FieldDefList

Points to the list of field definitions for the dataset.

Syntax:

```
property FieldDefList: TFieldDefList;
```

Description:

FieldDefList points to an internal list of field definitions for the dataset. **FieldDefList** represents a flattened view of the data, meaning that object fields in the dataset may be represented by several simple field definitions that represent the constituents of the object field. To determine the definitions in a hierarchical view, use **FieldDefs** instead.

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

5.1.19. FieldList

Lists the field components of a dataset.

Syntax:

```
property FieldList: TFieldList;
```

Description:

FieldList contains the names of all field components in the dataset. The fields are stored sequentially, or flattened out, meaning any child fields of an object field are stored as siblings in the **TFieldList.Fields** array.

5.1.20. Fields

Lists all non-aggregate field components of the dataset.

Syntax:

```
property Fields: TFields;
```

Description:

Use **Fields** to access field components. If fields are generated dynamically at runtime, the order of field components in **Fields** corresponds directly to the order of columns in the table or tables underlying a dataset. If a dataset uses persistent fields, then the order of field components corresponds to the ordering of fields specified in the **Fields editor** at design time.

When **ObjectView** is **True**, the fields are stored hierarchically, meaning any child fields of an object field are referenced by the object field and don't appear sequentially after the object field in the **TFields.Fields** array. When **ObjectView** is **False**, the fields are stored sequentially, or flattened out, meaning any child fields of an object field are stored sequentially in the **TFields.Fields** array.

Accessing fields with the **Fields** property is useful for applications that:

- Iterate over some or all fields in a dataset.
- Work with underlying tables whose internal data structure is unknown at runtime.

If an application knows the data types of individual fields, then it can read or write individual field values through the **Fields** property. For example, the following statement assigns a field value to the **Text** property of an edit box:

```
Edit1.Text := CustMySQLTable.Fields.Fields[6].AsString;
```

The preferred method for retrieving and assigning field values is to use persistent fields or the **FieldByName** method.

The following statements assign a value from an edit box to a field:

```
CusTMySQLTable.Edit;
CusTMySQLTable.Fields.Fields[6].AsString := Edit1.Text;
CusTMySQLTable.Post;
```

See also: Example: FieldCount,Fields,FieldName

5.1.21. FieldValues

Provides access to the values for all fields in the active record for the dataset.

Syntax:

```
property FieldValues[const FieldName: String]: Variant; default;
```

Description:

Use **FieldValues** to read and write values for fields in a dataset. **FieldName** is the name of a field to read from or write to.

FieldValues reads from and writes to fields whether **FieldName** represents simple field names, qualified field names for subfields of an object field. Because of this flexibility, it is often preferable to use the **FieldValues** property (or the **FieldByName** method) rather than the **Fields**, **FieldList** properties, all of which present a more limited selection of the dataset's fields.

FieldValues accepts and returns a *Variant*, so it can handle and convert fields of any type. Because **FieldValues** is the default property for **TDataSet**, you can omit the property name when referencing this property. For example, the following statements are semantically identical and write the value from an edit box into an integer field:

```
Customers.FieldValues['CustNo'] := Edit1.Text;
Customers['CustNo'] := Edit1.Text;
```

The following statement reads a string value from a field into an edit box:

```
Edit1.Text := Customers['Company'];
```

Because **FieldValues** always uses *Variants*, it may be a somewhat slower method of accessing data, than using a field native format (i.e., using a field b property), especially in applications that process large amounts of data.

See also: **Example: Append, FieldValues, Post**

5.1.22. Filter

Specifies the text of the current filter for a dataset.

Syntax:

```
property Filter: String;
```

Description:

Use **Filter** to specify a dataset filter. When filtering is applied to a dataset, only those records that meet a filter's conditions are available to an application. **Filter** contains the string that describes the filter condition.

For example, the following filter condition displays only those records where the State field is *'CA'* or *'MA'*:

```
State = 'CA' or State = 'MA'
```

Since DAC for MySQL v2.6.3 Filter property supports extended syntax:

- LIKE operator support.
- Sets of characters inside square brackets, e.g. [azAz], [a-dA-D].
- Exclamation sign (**!**) inside set indicates that any character should be matched except for those in set.
- Wildcard '**%**' matches any number of characters.
- A question mark '**?**' matches a single arbitrary character.

For example:

```
State LIKE '[mt]%' - will give us Montana, Texas, Michigan etc.
State LIKE '[!mt]%' - will give us all except states where first letter is 'm' or
't'
```

Applications can set **Filter** at runtime to change the filtering condition for a dataset at (for example, in response to user input).

5.1.23. Filtered

Specifies whether filtering is active for a dataset.

Syntax:

```
property Filtered: Boolean;
```

Description:

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

When filtering is enabled, user edits to a record may mean that the record no longer meets a filter's test condition. The next time the record is retrieved from the dataset while the filter is in effect, the record may seem to disappear. If that happens, the next record that passes the filter condition becomes the current record.

5.1.24. FilterOptions

Specifies whether filtering is case insensitive, and whether or not partial comparisons are permitted when filtering records.

Syntax:

```
property FilterOptions: TFilterOptions;
```

Description:

Set **FilterOptions** to specify whether or not filtering is case insensitive when filtering on string or character fields, and whether or not partial comparisons for matching filter conditions is allowed.

By default, **FilterOptions** is set to an empty set. For filters based on string fields, include **foCaseInsensitive** in **FilterOptions** to catch all variations on a string regardless of capitalization.

To prevent partial string comparisons, include **foNoPartialCompare** in **FilterOptions**.

> To filter strings bases on partial comparisons, exclude **foNoPartialCompare** from **FilterOptions** and use an asterisk as a wildcard.

For example:

```
State = 'M*'
```

5.1.25. Found

Indicates whether or not moving to a different record is successful.

Syntax:

```
property Found: Boolean;
```

Description:

Check the status of **Found** to determine if a call to **FindFirst**, **FindLast**, **FindNext** or, **FindPrior** succeeds. If **Found** is **True**, success is indicated. If **False**, the move to a different record failed.

5.1.26. KeySize

Specifies the size of the key for the current index of the dataset.

Syntax:

```
property KeySize: Word;
```

Description:

Check **KeySize** to determine the size, in bytes, of the key for the current index. **KeySize** varies depending on the number and type of fields that make up the current index.

5.1.27. LastInsertID

Get last inserted value of AUTO_INCREMENT column from MySQL server.

Syntax:

```
property LastInsertID : Int64;
```

Description:

This property uses TMySQLDatabase.LastInsertID to get value. DAC for MySQL performs 'SELECT LAST_INSERT_ID' query for current database and returns its result as value of this property.

See also: TMySQLDatabase.LastInsertID property

5.1.28. Modified

Indicates whether the active record is modified.

Syntax:

```
property Modified: Boolean;
```

Description:

Check **Modified** to determine if the active record is modified. If **Modified** is **True**, the active record is modified. If **False**, the active record is not modified.

In general, an application need not check the status of **Modified**. Properties, events, and methods of **TDataSet** and its descendants that modify records generally check this status automatically and take appropriate actions based on its value.

5.1.29. MultiResultsetNo

Since v2.5.3

Specifies the resultset to associate with component when it become active. This property is useful when query or stored procedure returns more than one dataset.

Syntax:

```
property MultiResultsetNo : integer
```

Description:

Set **MultiResultsetNo** to number of resultset you want to get from query or stored procedure that returns more that one result set. **EmySQLException** is raised if this value is less then number of resultsets returned by server. You can use AvailableResultsetCount property to inspect how many resultsets are returned by query or stored procedure.

Numbers of resultsets are counted from zero. So first resultset has number **0**, second has number **1**, ..., latest has number AvailableResultsetCount - 1

See also: AvailableResultsetCount property

5.1.30. Name

Designates the name of the dataset as referenced by other components.

Syntax:

```
property Name: TComponentName;
```

Description:

Use **Name** to change the name of a dataset to reflect its purpose in the current application. By default, the IDE assigns sequential names based on the type of the component, such as *'TMySQLTable1'*, *'TMySQLTable2'*, and so on.

When the name of the dataset is changed at design time, if any of the field components use the dataset name as a prefix to the field name, these field names are updated as well.

5.1.31. ObjectView

Specifies whether fields are to be stored hierarchically or flattened out in the **Fields** property.

Syntax:

```
property ObjectView: Boolean;
```

Description:

ObjectView affects the way the **Fields** property stores object fields and the way data-aware grids display ADT and array fields.

When **ObjectView** is **True**, the fields are stored hierarchically in the **Fields** property, meaning any child fields of an object field are referenced by the object field and don't appear sequentially after the object field in the **TFields.Fields** array. When **ObjectView** is **False**, the fields are stored sequentially in the **Fields** property, meaning any child fields of an object field are stored after the object field as siblings in the **Fields** array.

When **ObjectView** is **False**, object field types, such as **TADTField**, are not created. This switch is provided for increased compatibility with older data-aware controls, which may not be able to handle object field types properly. The default is **False** for **TMySQLDataSet**.

5.1.32. RecNo

Indicates the current record in the dataset.

Syntax:

```
property RecNo: Longint;
```

Description:

Examine **RecNo** to determine the record number of the current record in the dataset. Applications might use this property with **RecordCount** to iterate through all the records in a dataset, though typically record iteration is handled with calls to **First**, **Last**, **MoveBy**, **Next**, and **Prior**.

> If accessing tables, **RecNo** can be set to a specific record number to position the cursor on that record, beginning from **0**.

5.1.33. RecordCount

Indicates the total number of records associated with the dataset.

Syntax:

```
property RecordCount: Longint;
```

Description:

Examine **RecordCount** to determine the total number of records in the dataset. Applications might use this property with **RecNo** to iterate through all the records in a dataset, though typically record iteration is handled with calls to **First**, **Last**, **MoveBy**, and **Prior**.

See also: Example: Min, Max, Position, RecordCount, First, Next

5.1.34. RecordSize

Indicates the size of a record in the dataset.

Syntax:

```
property RecordSize: Word;
```

Description:

Examine **RecordSize** to determine the physical size, in bytes, of the buffer Delphi allocates to hold a record in the dataset. When a dataset is opened, the **Open** procedure requests record-buffer size information and stores the returned information in **RecordSize**. Delphi uses this information internally. Applications seldom, if ever, require this information.

5.1.35. RefreshDelete

Specifies whether a dataset should delete a record locally when a database record is not found at refreshing.

Syntax:

```
property RefreshDelete : Boolean default True;
```

Description:

Use the **RefreshDelete** property to allow a dataset to delete a record from a local storage, when a corresponding database record is not found at **RefreshRecord** call. The default value is **True**. The

most probable reason why the record is not found - it is deleted. There may be other reasons as well. To delete the record, set this property to **True**. To leave the record and raise an exception, set this property to False. For additional details, see the description of the **RefreshRecord** method.

5.1.36. SortFieldNames

Specifies field names and sorting order to sort opened dataset by these fields on the client side without refetching data from server.

Syntax:

```
property SortFieldNames : string;
```

Description:

Set **SortFieldNames** to establish or change the list of fields on which the dataset is sorted. Set sort to the name of a single field or to a comma-separated list of fields. Every field name can be followed by the keyword *'ASC'* or *'DESC'* to specify a sort direction for the field. If one of these keywords is not used, the default sort direction for the field is ascending (*'ASC'*).

For example:

```
mySQLQuery1.SortFieldNames := 'ID, Name DESC, ColorValue ASC';
```

Since v2.6.3 double-quote character (") can be used to quote field name if one contain spaces, commas or other non-alphanumeric character.

For example:

```
mySQLQuery1.SQL.Clear;
mySQLQuery1.SQL.Add('SELECT LEFT(TABLE_NAME, 20), TABLE_COLLATION FROM
INFORMATION_SCHEMA.TABLES');
mySQLQuery1.Open;
mySQLQuery1.SortFieldNames := 'TABLE_COLLATION, "LEFT(TABLE_NAME, 20)" DESC';
```

If dataset is opened setting this property to some string causes sorting of dataset immediately. If dataset is closed (**Active** = **False**) it will be sorted by this fields after opening. This property can also be used at Design-time.

> Since v2.7.6 you can adjust case sensitivity of sorting using TMySQLDatabase.DatasetOptions property.

Example:

This code can be used to sort data in **TDBGrid** component by particular column when user clicks on its title.

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);
begin
  //if column already sorted lets sort it in reverse order
  if mySQLTable1.SortFieldNames = Column.FieldName then
    mySQLTable1.SortFieldNames := Column.FieldName + ' DESC'
  else
    mySQLTable1.SortFieldNames := Column.FieldName;
end;
```

See also: SortBy method, TMySQLDatabase.DatasetOptions property

5.1.37. SparseArrays

Determines whether a unique **TField** object is created for each element of an array field.

Syntax:

```
property SparseArrays: Boolean;
```

Description:

When opening a table or client data set with an array field, **SparseArrays** determines whether a unique field component is created for each element of the array field.

The default is **False**, where a **TField** descendant is created for each element of the array field. If you plan to bind data-aware controls to elements of the array field, **SparseArrays** must be set to **False**.

Setting **SparseArrays** to **True** conserves memory, but does not allow applications to gain a reference to an element of the array field.

5.1.38. State

Indicates the current operating mode of the dataset.

Syntax:

```
property State: TDataSetState;
```

Description:

Examine **State** to determine the current operating mode of the dataset. **State** determines what can be done with data in a dataset, such as editing existing records or inserting new ones. The dataset state constantly changes as an application processes data.

Opening a dataset changes **State** from **dsInactive** to **dsBrowse**. An application can call **Edit** to put a dataset into **dsEdit** state, or call **Insert** to put a dataset into **dsInsert** state. If a dataset is a **TMySQLTable** object, an application can call **SetKey** or **SetRange** to put the dataset into **dsSetKey** state.

Posting or canceling edits, insertions, or deletions, changes **State** from its current state to **dsBrowse**. Closing a dataset changes its state to **dsInactive**.

Some states, such as **dsCalcFields**, **dsFilter**, **dsNewValue**, **dsOldValue**, and **dsCurValue** cannot be seen or set directly by an application. These states are automatically set when **OnCalcField** and **OnFilterRecord** events occur, or when an application accesses certain field properties.

See also: Example: State,Seek,Truncate

5.1.39. StatementID

Return the statement ID of the prepared query.

Syntax:

```
property StatementID : integer;
```

Description:

Use **StatementID** property to determinate the statement id of previously prepared query.

See also: Prepare, UnPrepare property

5.1.40. UpdateMode

Determines how MySQL finds records when updating to an SQL database.

Syntax:

```
property UpdateMode: TUpdateMode;
```

Description:

Use **UpdateMode** to specify the criteria to use when locating a record in the dataset. **UpdateMode** specifies whether modified records are located based on all columns (fields), on only the key fields, or on the key fields plus the original values of fields that have been modified.

5.1.41. UpdateObject

Specifies the update object component used to update a read-only result set.

Syntax:

```
property UpdateObject: TDataSetUpdateObject;
```

Description:

Use **UpdateObject** to specify the TMySQLUpdateSQL component to use in an application that must be able to update a read-only result set.

MySQL always attempts to provide an updatable, or "live" query result unless an application specifically requests a read-only view of data. In some cases, such as a query made against multiple tables, a live result set cannot be returned. In these cases, **UpdateObject** property can be used to specify a TMySQLUpdateSQL component that performs updates as a separate transaction that is transparent to the application.

See also: TMySQLUpdateSQL component

5.2. Methods

Please see TMySQLDataset methods short descriptions below:

Derived from TDataSet

ActiveBuffer
Returns a pointer to the buffer for the active record.

Append
Adds a new, empty record to the end of the dataset.

AppendRecord
Adds a new, populated record to the end of the dataset and posts it to the database.

CheckBrowseMode
Automatically posts or cancels data changes when an application changes which record in the dataset is the active record.

ClearFields
Clears the contents of all fields for the active record.

Close
Closes a dataset.

ControlsDisabled
Indicates whether data-aware controls do not update their display to reflect changes to the dataset.

CursorPosChanged
Marks the internal cursor position as invalid.

Delete
Deletes the active record and positions the cursor on the next record.

DisableControls
Disables data display in data-aware controls associated with the dataset.

Edit
Enables editing of data in the dataset.

EnableControls
Re-enables data display in data-aware controls associated with the dataset.

FieldByName
Finds a field based on its name.

FindField
Searches for a specified field in the dataset.

FindFirst
Implements a virtual method for positioning the cursor on the first record in a filtered dataset.

FindLast
Implements a virtual method for positioning the cursor on the last record in a filtered dataset.

FindNext
Implements a virtual method for positioning the cursor on the next record in a filtered dataset.

FindPrior
Implements a virtual method for positioning the cursor on the previous record in a filtered dataset.

First
Positions the cursor on the first record in the dataset.

FreeBookmark
Frees the resources allocated for a specified bookmark.

GetBookmark
Allocates a bookmark for the current cursor position in the dataset.

GetDetailDataSets

Fills a list with a dataset for every detail dataset that is not the value of a nested dataset field.

GetFieldList
Retrieves a specified set of field objects into a list.

GetFieldNames
Retrieves a list of names for all fields in a dataset.

GotoBookmark
Implements a virtual method to position the cursor on the record pointed to by a specified bookmark.

Insert
Inserts a new, empty record in the dataset.

InsertRecord
Inserts a new, populated record to the dataset and posts it to the database.

IsEmpty
Indicates whether the dataset contains no records.

IsLinkedTo
Indicates whether a dataset is linked to a specified data source.

Last
Positions the cursor on the last record in the dataset.

MoveBy
Positions the cursor on a record relative to the active record in the dataset.

Next
Positions the cursor on the next record in the dataset.

Open
Opens the dataset.

Prior

Positions the cursor on the previous record in the dataset.

Refresh
Refetches data from the database to update a dataset's view of data.

Resync
Refetches the active record and the records that precede and follow it.

SetFields
Sets the values for all fields in a record.

UpdateCursorPos
Positions the cursor on the active record.

UpdateRecord
Ensures that data-aware controls and detail datasets reflect record updates.

In TMySQLDataSet

ApplyUpdates
Writes a dataset's pending cached updates to the database.

BookmarkValid
Tests the validity of a specified bookmark.

Cancel
Cancels modifications to the current record if those changes are not yet posted.

CancelUpdates
Clears all pending cached updates from the cache and restores the dataset its prior state.

CheckOpen
Checks the result of a call to the MySQL.

CloseDatabase
Closes a database connection associated with the database.

CommitUpdates
Clears the cached updates buffer.

CompareBookmarks
Indicates the relationship between two bookmarks.

FetchAll
Retrieves all records from the current cursor position to the end of the file and stores them locally.

FlushBuffers
Posts all changes that have been written to the record buffer.

GetBlobFieldData
Reads BLOB data into a buffer.

GetCurrentRecord
Retrieves the current record into a buffer.

GetFieldData
Retrieves the current value of a field into a buffer.

GetFieldType
Retrieves a internal field types defined in the DAC for MySQL modules.

GetIndexInfo
Retrieves information about the current index into the index data fields of the dataset.

GetLastInsertID
Returns the ID generated for an AUTO_INCREMENT column by the previous query.

Locate

Searches the dataset for a specified record and makes that record the current record.

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

OpenDatabase
Opens the database that contains the dataset.

Post
Writes a modified record to the database.

Post
Writes a modified record to the database.

Prepare
Sends a query for optimization prior to execution.

UnPrepare
Frees the resources allocated for a previously prepared query.

RefreshRecord
Rereads the field values of the current record from a data source.

RevertRecord
Restores the current record in the dataset to an unmodified state when cached updates are enabled.

SortBy
Sorts opened dataset on client side without refetching data from server.

Translate
Converts a data string between the ANSI character set used by Delphi (and Windows), and the local code page (OEM character set).

UpdateStatus
Reports the update status for the current record.

5.2.1. ActiveBuffer

Returns a pointer to the buffer for the active record.

Syntax:

```
function ActiveBuffer: PChar;
```

Description:

ActiveBuffer is used internally by many dataset methods to ensure that the active buffer points to the buffer for the active record. If an application uses existing dataset methods, the active buffer is always set correctly, so there is usually no need to call **ActiveBuffer** directly.

ActiveBuffer is also used by bookmarking methods to index into the active record buffer to retrieve bookmark information.

Applications that provide custom dataset routines may need to call **ActiveBuffer** to access the buffer data.

5.2.2. Append

Adds a new, empty record to the end of the dataset.

Syntax:

```
procedure Append;
```

Description:

Call **Append** to:

- Open a new, empty record at the end of the dataset.
- Set the active record to the new record.

After a call to **Append**, an application can enable users to enter data in the fields of the record, and can then post those changes to the database using **Post** (or **ApplyUpdates** if cached updating is enabled).

See also: Example: Append, FieldValues, Post

5.2.3. AppendRecord

Adds a new, populated record to the end of the dataset and posts it to the database.

Syntax:

```
procedure AppendRecord(const Values: array of const);
```

Description:

Call **AppendRecord** to create a new, empty record at the end of the dataset, populate it with the field values in **Values**, and post the values to the database.

For MySQL indexed tables, the index is updated with the new record information. The newly appended record becomes the active record.

Example:

This statement appends a record to the *Customer* table. Note that *Nulls* are entered for some of the values, but are not required for missing values at the end of the array argument, i.e. after the *Discount* field.

```
Customer.AppendRecord([CustNoEdit.Text,
                       CoNameEdit.Text,
                       AddrEdit.Text,
                       Null,
                       Null,
                       Null,
                       Null,
                       Null,
                       Null,
                       DiscountEdit.Text]);
```

5.2.4. ApplyUpdates

Writes a dataset's pending cached updates to the database.

Syntax:

```
procedure ApplyUpdates;
```

Description:

Call **ApplyUpdates** to write a dataset's pending cached updates to a database. This method passes cached data to the database for storage, but the changes are not committed to the database. An application must explicitly call the database component's **Commit** method to commit the changes to the database if the write is successful, or call the database's **Rollback** method to undo the changes if there is an error.

Following a successful write to the database, and following a successful call to the database's **Commit** method, an application should call the **CommitUpdates** method to clear the cached update buffer.

> The preferred method for updating datasets is to call a database component's **ApplyUpdates** method rather than to call each individual dataset's **ApplyUpdates** method. The database component's **ApplyUpdates** method takes care of committing and rolling back transactions and clearing the cache when the operation is successful.

5.2.5. BookmarkValid

Tests the validity of a specified bookmark.

Syntax:

```
function BookmarkValid(Bookmark: TBookmark): Boolean; override;
```

Description:

Call **BookmarkValid** to determine if a specified bookmark is currently assigned a value. **Bookmark** specifies the bookmark to test.

BookmarkValid returns **True** if a bookmark is valid. Otherwise, it returns **False**.

5.2.6. Cancel

Cancels modifications to the current record if those changes are not yet posted.

Syntax:

```
procedure Cancel;
```

Description:

Call **Cancel** to undo modifications made to one or more fields belonging to the current record. As long as those changes are not already posted to the database, **Cancel** returns the record to its previous state, and sets the dataset state to **dsBrowse**.

Typically **Cancel** is used to back out of changes in response to user request, or in field validation routines that back out illegal field values. The **TDBNavigator** object contains a **Cancel** button that triggers a call to **Cancel**.

Example:

The following fragment prompts the user to confirm changes to a record; if the user clicks **Yes**, the record is posted to the table, otherwise the changes are canceled.

```
if MessageDlg('Update Record?',
               mtConfirmation,
               [mbYes, mbNo], 0) = mrYes then
  MySQLTable1.Post
else
  MySQLTable1.Cancel;
```

5.2.7. CancelUpdates

Clears all pending cached updates from the cache and restores the dataset its prior state.

Syntax:

```
procedure CancelUpdates;
```

Description:

Call **CancelUpdates** to clear all pending cached updates from the cache and restore the dataset to the state it was in when the table was opened, cached updates were last enabled, or updates were last successfully applied to the database.

When a dataset is closed, or the **CachedUpdates** property is set to **False**, **CancelUpdates** is called automatically.

To undo changes to a single record, call **RevertRecord**.

5.2.8. CheckBrowseMode

Automatically posts or cancels data changes when an application changes which record in the dataset is the active record.

Syntax:

```
procedure CheckBrowseMode;
```

Description:

CheckBrowseMode is used internally by many dataset methods to ensure that modifications to the active record are posted to the database when a dataset's state is **dsEdit**, **dsInsert**, or **dsSetKey** state and a method switches to a different record.

- If **State** is **dsEdit** or **dsInsert**, **CheckBrowseMode** calls **UpdateRecord**, and then, if the **Modified** property for the dataset is **True**, calls **Post**. If **Modified** is **False**, **CheckBrowseMode** calls **Cancel**.
- If **State** is **dsSetKey**, **CheckBrowseMode** calls **Post**.
- If **State** is **dsInactive**, **CheckBrowseMode** raises an exception.
- If an application uses existing dataset methods, **CheckBrowseMode** is always called when necessary, so there is usually no need to call **CheckBrowseMode** directly.

Applications that provide custom dataset routines may need to call **CheckBrowseMode** inside those routines to guarantee that changes are posted to the database when switching to a different record.

5.2.9. CheckOpen

Checks the result of a call to the MySQL.

Syntax:

```
function CheckOpen(Status: DBIResult): Boolean;
```

Description:

Call **CheckOpen** to determine if a call to the MySQL returns an error when an attempt is made to access a dataset. **Status** is the return result of a previous call.

CheckOpen returns **True** if access is successful. If **Status** indicates insufficient table rights when accessing a table, **CheckOpen** calls the **GetPassword** method to prompt the user for a password. If the dialog is successful, **CheckOpen** returns **True**.

Otherwise **CheckOpen** returns **False**, indicating that dataset access failed.

5.2.10. ClearFields

Clears the contents of all fields for the active record.

Syntax:

```
procedure ClearFields;
```

Description:

Call **ClearFields** to erase the current contents of all fields for the active record. If the dataset is not in either **dsInsert** or **dsEdit** state, **ClearFields** raises an exception.

If a **SetKey** operation is not under way, **ClearFields** recalculates all calculated fields, and generates an **OnDataChange** event on the data source component associated with the dataset.

5.2.11. Close

Closes a dataset.

Syntax:

```
procedure Close;
```

Description:

Call **Close** to set the **Active** property of a dataset to **False**. When **Active** is **False**, the dataset is closed; it cannot read data from or write data to the database.

An application must set **Active** to **False** before changing other properties that affect the status of the database or the controls that display data in an application.

For example, to change the **DataSource** property for a dataset, the dataset must be closed. Closing the dataset puts it into the **dsInactive** state and closes the cursor.

5.2.12. CloseDatabase

Closes a database connection associated with the database.

Syntax:

```
procedure CloseDatabase(Database: TMySQLDataBase);
```

Description:

Call **CloseDatabase** to close a persistent database connection. **Database** specifies the database component for which to close the connection.

CloseDatabase decrements the specified database component's reference count, and then, if the reference count is zero and the database component's **KeepConnection** property is **False**, **CloseDatabase** either frees a temporary database component or closes the connection for a persistent database component.

> Calling **CloseDatabase** for a persistent database component does not close the connection. To close a connection for a persistent database component, call the database component's **Close** method directly.

Temporary database components are closed automatically when the last dataset associated with the database component is closed, but an application can call **CloseDatabase** prior to that time to force closure. Closing a connection established by a temporary database component does not free the component if the **KeepConnection** property is **True** (the default). To free temporary database components after closing their connections call **DropConnections** method.

5.2.13. CommitUpdates

Clears the cached updates buffer.

Syntax:

```
procedure CommitUpdates;
```

Description:

Call **CommitUpdates** to clear the cached updates buffer after both a successful call to **ApplyUpdates** and a database component's **Commit** method. Clearing the cache after applying updates ensures that the cache is empty except for records that could not be processed and were skipped by the **OnUpdateRecord** or **OnUpdateError** event handlers. An application can attempt to modify the records still in the cache.

Record modifications made after a call to **CommitUpdates** repopulate the cached update buffer and require a subsequent call to **ApplyUpdates** to move them to the database.

> Applications that use a database component's **ApplyUpdates** method to apply and commit pending updates for all datasets associated with the database component do not need to call **CommitUpdates**.

5.2.14. CompareBookmarks

Indicates the relationship between two bookmarks.

Syntax:

```
function CompareBookmarks(Bookmark1,
                          Bookmark2: TBookmark): Integer;
```

Description:

Call **CompareBookmarks** to determine if two bookmarks are identical. *Bookmark1* and *Bookmark2* are the bookmarks to compare. **CompareBookmarks** returns **-1** if *Bookmark1* is less than *Bookmark2*, **1** if *Bookmark1* is greater than *Bookmark2*, and **0** if the bookmarks are identical or **nil**.

5.2.15. ControlsDisabled

Indicates whether data-aware controls do not update their display to reflect changes to the dataset.

Syntax:

```
function ControlsDisabled: Boolean;
```

Description:

Call **ControlsDisabled** to determine if the updating of data display in data-aware controls is currently disabled. If **ControlsDisabled** is **True**, controls are currently disabled. **ControlsDisabled** is **True** as long as the reference count that keeps track of disabling for the dataset is greater than zero. This count is incremented every time the **DisableControls** procedure is called and decremented when **EnableControls** is called. Applications should call **DisableControls** to improve performance and prevent constant updates during automated iterations through records in the dataset.

In complex applications, when controls may be disabled multiple times by different processes, you can use **ControlsDisabled** as a check in a procedure to re-enable controls should each call to **DisableControls** not be paired with a subsequent call to **EnableControls**.

Example:

```
procedure ReEnableControls (DataSet: TDataSet);
begin
while DataSet.ControlsDisabled do
  DataSet.EnableControls;
end;
```

5.2.16. CursorPosChanged

Marks the internal cursor position as invalid.

Syntax:

```
procedure CursorPosChanged;
```

Description:

CursorPosChanged is an internal method that invalidates the variable that tracks the physical cursor position relative to the logical cursor position. **CursorPosChanged** is called by the **Locate** and **Lookup** methods prior to searching for a requested record. These methods, if successful, reposition the cursor at the first matching record found.

5.2.17. Delete

Deletes the active record and positions the cursor on the next record.

Syntax:

```
procedure Delete;
```

Description:

Call **Delete** to remove the active record from the database. If the dataset is inactive, **Delete** raises an exception.

Otherwise **Delete**:

- Verifies that the dataset is not empty (and raises an exception if it is).
- Calls **CheckBrowseMode** to post any pending changes to a prior record if necessary.
- Calls the **BeforeDelete** event handler.
- Deletes the record.
- Frees the buffers allocated for the record.
- Puts the dataset into **dsBrowse** mode.
- Resynchronizes the dataset to position the cursor on the next undeleted record.
- Calls the **AfterDelete** event handler.

5.2.18. DisableControls

Disables data display in data-aware controls associated with the dataset.

Syntax:

```
procedure DisableControls;
```

Description:

Call **DisableControls** prior to iterating through a large number of records in the dataset to prevent data-aware controls from updating every time the active record changes. Disabling controls prevents flicker and speeds performance because data does not need to be written to the display.

If controls are not already disabled, **DisableControls** records the current state of the dataset, broadcasts the state change to all associated data-aware controls and detail datasets, and increments the dataset's disabled count variable. Otherwise, **DisableControls** just increments the disabled count variable.

The disabled count is used internally by other methods and objects to determine whether to display data in data-aware controls. When the disable count variable is greater than zero, data is not displayed.

If the dataset is the master of a master/detail relationship, calling **DisableControls** also disables the master/detail relationship. Setting **BlockReadSize** instead of calling **DisableControls** updates the detail datasets as you scroll through the dataset, but does not update data-aware controls.

> **DisableControls** can safely be called when controls are already disabled. In complex applications there may be separate operations that are sometimes nested, both of which need to disable controls.

See also: Example: DisableControls, EnableControls, Eof

5.2.19. Edit

Enables editing of data in the dataset.

Syntax:

```
procedure Edit;
```

Description:

Call **Edit** to permit editing of the active record in a dataset. **Edit** determines the current state of the dataset. If the dataset is empty, **Edit** calls **Insert**.

Otherwise **Edit**:

- Calls **CheckBrowseMode** to post any pending changes to a prior record if necessary.
- Calls the **BeforeEdit** event handler.
- Retrieves the record.
- Puts the dataset into **dsEdit** state, enabling the application or user to modify fields in the record.
- Broadcasts the state change to associated controls.
- Calls the **AfterEdit** event handler.

See also: Example: Create, CreateBlobStream, Edit, CopyFrom

5.2.20. EnableControls

Re-enables data display in data-aware controls associated with the dataset.

Syntax:

```
procedure EnableControls;
```

Description:

Call **EnableControls** to permit data display in data-aware controls after a prior call to **DisableControls**.

EnableControls decrements the disabled count variable for the dataset if it is not already zero. If the disable count variable is zero, **EnableControls** updates the current state of the dataset, if necessary, and then tells associated controls to re-enable display.

See also: Example: DisableControls, EnableControls, Eof

5.2.21. FetchAll

Retrieves all records from the current cursor position to the end of the file and stores them locally.

Syntax:

```
procedure FetchAll;
```

Description:

Call **FetchAll** to reduce network traffic when using cached updates. **FetchAll** calls **CheckBrowseMode** to post any pending changes, and then retrieves all records from the current cursor position to the end of the file, and store them locally.

Ordinarily when cached updates are enabled, a transaction retrieves only as much data as it needs for display purposes. Each new fetch starts a new, read-only transaction. To consolidate transactions and reduce network traffic, an application can call **FetchAll** to retrieve all data in a single transaction.

Using **FetchAll** is not always appropriate. For example, when an application accesses a database used by many simultaneous clients and there is a high degree of contention for updating the same records, fetching all records at once may not be advantageous because some fetched records may be changed by other applications. Always weigh the advantages of reduced network traffic against the need for reduced record contention.

5.2.22. FieldByName

Finds a field based on its name.

Syntax:

```
function FieldByName(const FieldName: String): TField;
```

Description:

Call **FieldByName** to retrieve field information for a field when only its name is known. **FieldName** is the name of an existing field. **FieldByName** returns the **TField** component for the specified field. If the specified field does not exist, **FieldByName** raises an **EDatabaseError** exception.

FieldName can be the name of a simple field, the name of a subfield of an object field that has been qualified by the parent field name, or the name of an aggregated field. Because of this flexibility, it is often preferable to use **FieldByName** rather than the Fields property or the **AggFields** property.

An application can directly access specific properties and methods of the field returned by **FieldByName**. For example, the following statement determines if a specified field is a calculated field or not:

```
if Customers.FieldByName('FullName').Calculated then
  MessageDlg(Format('%s is a calculated field. ',
                    ['FullName']),
                    mtInformation,
                    [mbOK],
                    0);
```

FieldByName is especially useful at design time for developers who are creating database applications, but who do not have access to the underlying table and therefore cannot use persistent field components.

To retrieve or set the value for a specific field, call the default dataset method **FieldValues** instead of **FieldByName**.

See also: Example: EditRangeStart,EditRangeEnd,FieldByName,ApplyRange

5.2.23. FindField

Searches for a specified field in the dataset.

Syntax:

```
function FindField(const FieldName: String): TField;
```

Description:

Call **FindField** to determine if a specified field component exists in a dataset. **FieldName** is the name of the field for which to search. If **FindField** finds a field with a matching name, it returns the **TField** component for the specified field. Otherwise it returns **nil**.

FindField is the same as the **FieldByName** method, except that it returns **nil** rather than raising an exception when the field is not found.

See also: Example: FindField,AsString

5.2.24. FindFirst

Implements a virtual method for positioning the cursor on the first record in a filtered dataset.

Syntax:

```
function FindFirst: Boolean;
```

Description:

This function returns **False**, indicating that the cursor was not successfully repositioned. Descendant classes override **FindFirst** to position the cursor on the first record of the dataset, honoring any filters that are in effect. **FindFirst** should return **True** if the cursor is successfully repositioned.

> **Delphi 7** and prior has poor support for *int64* values in *variant* type. This means that you'll be unable to use Locate and similar methods with such fields. Lookup fields and master-detail tables will not work properly too because of their dependence on **Locate** method. Please update your IDE to BDS 2006 (or later) or don't use BIGINT and UNSIGNED INT datatypes for keys and indexes if you need to use them in lookup fields. Also you can't use variant-style properties (**FieldValues**, **AsVariant** and so on) with such fields.

5.2.25. FindLast

Implements a virtual method for positioning the cursor on the last record in a filtered dataset.

Syntax:

```
function FindLast: Boolean;
```

Description:

This function returns **False**, indicating that the cursor was not successfully repositioned. Descendant classes override **FindLast** to position the cursor on the last record of the dataset, honoring any filters that are in effect. **FindLast** should return **True** if the cursor is successfully repositioned.

> **Delphi 7** and prior has poor support for *int64* values in *variant* type. This means that you'll be unable to use Locate and similar methods with such fields. Lookup fields and master-detail tables will not work properly too because of their dependence on **Locate** method. Please update your IDE to BDS 2006 (or later) or don't use BIGINT and UNSIGNED INT datatypes for keys and indexes if you need to use them in lookup fields. Also you can't use variant-style properties (**FieldValues**, **AsVariant** and so on) with such fields.

5.2.26. FindNext

Implements a virtual method for positioning the cursor on the next record in a filtered dataset.

Syntax:

```
function FindNext: Boolean;
```

Description:

This function returns **False**, indicating that the cursor was not successfully repositioned. Descendant classes override **FindNext** to position the cursor on the next record of the dataset, honoring any filters that are in effect. **FindNext** should return **True** if the cursor is successfully repositioned.

> **Delphi 7** and prior has poor support for *int64* values in *variant* type. This means that you'll be unable to use Locate and similar methods with such fields. Lookup fields and master-detail tables will not work properly too because of their dependence on **Locate** method. Please update your IDE to BDS 2006 (or later) or don't use BIGINT and UNSIGNED INT datatypes for keys and indexes if you need to use them in lookup fields. Also you can't use variant-style properties (**FieldValues**, **AsVariant** and so on) with such fields.

5.2.27. FindPrior

Implements a virtual method for positioning the cursor on the previous record in a filtered dataset.

Syntax:

```
function FindPrior: Boolean;
```

Description:

This function returns **False**, indicating that the cursor was not successfully repositioned. Descendant classes override **FindPrior** to position the cursor on the previous record of the dataset, honoring any filters that are in effect. **FindPrior** should return **True** if the cursor is successfully repositioned.

> **Delphi 7** and prior has poor support for *int64* values in *variant* type. This means that you'll be unable to use Locate and similar methods with such fields. Lookup fields and master-detail tables will not work properly too because of their dependence on **Locate** method. Please update your IDE to BDS 2006 (or later) or don't use BIGINT and UNSIGNED INT datatypes for keys and indexes if you need to use them in lookup fields. Also you can't use variant-style properties (**FieldValues**, **AsVariant** and so on) with such fields.

See also: Example: GetBookmark, GotoBookmark, FreeBookmark, FindPrior, Value, OnDataChange, BOF

5.2.28. First

Positions the cursor on the first record in the dataset.

Syntax:

```
procedure First;
```

Description:

Call **First** to position the cursor on the first record in the dataset and make it the active record. **First** posts any changes to the active record and:

- Clears the record buffers.
- Sets the cursor to the beginning of the dataset.
- Fetches the first record, positions the cursor on it, and makes it the active record.
- Fetches any additional records required for display, such as those needed to fill out a grid control.
- Sets the **Bof** property to **True**.
- Broadcasts the record change so that data controls and linked detail sets can update.

> **TDataSet** uses internal, protected methods to position the database cursor and to fetch additional records required for display. In **TDataSet**, these internal methods are abstract. Descendant classes implement these methods to enable the First method to work.

See also: Example: Min, Max, Position, RecordCount, First, Next

5.2.29. FlushBuffers

Posts all changes that have been written to the record buffer.

Syntax:

```
procedure FlushBuffers;
```

Description:

Call **FlushBuffers** to cause the dataset to post all pending changes to the database, including any cached updates. Use **FlushBuffers** instead of **CheckBrowseMode** if it is important that cached record buffers are posted.

5.2.30. FreeBookmark

Frees the resources allocated for a specified bookmark.

Syntax:

```
procedure FreeBookmark(Bookmark: TBookmark); virtual;
```

Description:

Call **FreeBookmark** to free an existing bookmark before reassigning it. **FreeBookmark** releases the memory allocated for a specified bookmark when the bookmark is no longer needed.

See also: Example: GetBookmark, GotoBookmark, FreeBookmark, FindPrior, Value, OnDataChange, BOF

5.2.31. GetBlobFieldData

Reads BLOB data into a buffer.

Syntax:

```
TBlobByteData = array of Byte;
function GetBlobFieldData(FieldNo: Integer;
```

```
                        var Buffer: TBlobByteData): Integer;
override;
```

Description:

GetBlobFieldData reads blob data from the field specified by **FieldNo** into a **Buffer**. The buffer is a dynamic array of bytes, so that it can grow to accommodate the size of the BLOB data. **GetBlobFieldData** returns the size of the buffer.

5.2.32. GetBookmark

Allocates a bookmark for the current cursor position in the dataset.

Syntax:

```
function GetBookmark: TBookmark; virtual;
```

Description:

Call **GetBookmark** to establish a bookmark for the active record in the dataset. Establishing a bookmark for a record enables an application to return to that record in the dataset at any time while the bookmark exists.

GetBookmark requires that a variable of type **TBookmark** already be declared in the application. Use **GetBookmark** to assign the variable a value that can be referenced by subsequent calls to **GotoBookmark** and **FreeBookmark**.

> Applications that create bookmarks with **GetBookmark** should subsequently release the system resource allocated to them by calling **FreeBookmark** when the bookmarks are no longer needed.

See also: Example: GetBookmark, GotoBookmark, FreeBookmark, FindPrior, Value, OnDataChange, BOF

5.2.33. GetCurrentRecord

Retrieves the current record into a buffer.

Syntax:

```
function GetCurrentRecord(Buffer: PChar): Boolean;
```

Description:

Most applications should not need to call **GetCurrentRecord**. **TDataSet** automatically allocates a buffer for the active record.

Call **GetCurrentRecord** to copy the current record into a buffer allocated by the application. **Buffer** must be at least as big as the record size indicated by the **RecordSize** property.

5.2.34. GetDetailDataSets

Fills a list with a dataset for every detail dataset that is not the value of a nested dataset field.

Syntax:

```
procedure GetDetailDataSets(List: TList);
```

Description:

Datasets can represent master/detail relationships in two ways: as linked cursors or as nested dataset fields. **GetDetailDataSets** lists all detail datasets of the active record into List if they are not the value of a nested dataset field. To obtain a list of the detail datasets that are the values of nested dataset fields, use the **NestedDataSets** property instead.

5.2.35. GetFieldData

Retrieves the current value of a field into a buffer.

Syntax:

```
function GetFieldData(FieldNo: Integer; Buffer: Pointer): Boolean; overload;
override;
function GetFieldData(Field: TField; Buffer: Pointer): Boolean; overload;
override;
function GetFieldData(Field: TField; Buffer: Pointer; NativeFormat: Boolean):
Boolean; overload; virtual;
```

Description:

Most applications do not need to call **GetFieldData**. **TField** objects call this method to implement their **GetData** method.

The **Field** or **FieldNo** parameter indicates the field whose data should be fetched. **Field** specifies the component itself, while **FieldNo** indicates its field number. The **Buffer** parameter is a memory buffer

with sufficient space to accept the value of the field as it exists in the database (unformatted and untranslated).

NativeFormat indicates whether the dataset fetches the field in Delphi's native format for the field type. When **NativeFormat** is **False**, the dataset must convert the field value to the native type. This allows the field to handle data from different types of datasets in a uniform manner.

GetFieldData returns a value that indicates whether the data was successfully fetched.

GetFieldData returns **True** if the buffer is successfully filled with the field data, and **False** if the data could not be fetched.

5.2.36. GetFieldList

Retrieves a specified set of field objects into a list.

Syntax:

```
procedure GetFieldList(List: TList; Const FieldNames: String);
```

Description:

Call **GetFieldList** to copy a specified set of field objects into a list object created and maintained by the application.

List is the **TList** object into which to copy the field objects. **FieldNames** is a string containing the name of the fields to copy. Each field name in the string must be separated from other field names with a semicolon. **GetFieldList** builds a list that contains only the field objects for which it finds a matching name in the dataset's list of field objects.

> Applications do not normally call **GetFieldList** to copy field objects. Field objects are directly accessible through the dataset itself. In some cases, however, it can be useful to work with a copy of a field object or its data instead of working on the actual object in the dataset.

5.2.37. GetFieldNames

Retrieves a list of names for all fields in a dataset.

Syntax:

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

Description:

Call **GetFieldNames** to get a list of names for all fields in a dataset. List is a **TStrings** object created and maintained by the application.

For example, to load a list box with the field names of a table:

```
MySQLTable1.GetFieldNames(ListBox1.Items);
```

Retrieving a list of field names is especially useful for applications that work with datasets whose field objects are created dynamically at runtime. By retrieving a list of field names, the application can be restricted to working only with fields that exist at runtime.

5.2.38. GetFieldType

Retrieves a internal field types defined in the DAC for MySQL modules.

Syntax:

```
function GetFieldType(const FieldNum: integer): integer;
```

Description:

The **FieldNum** parameter indicates the field whose type identifier should be fetched. **FieldNum** indicates its field number.

5.2.39. GetIndexInfo

Retrieves information about the current index into the index data fields of the dataset.

Syntax:

```
procedure GetIndexInfo;
```

Description:

Call **GetIndexInfo** to update information about the current index. Ordinarily an application does not need to call **GetIndexInfo** because this routine is used internally to retrieve index information as needed. Some applications, however, may want to ensure that the index information used by the dataset is up-to-date.

GetIndexInfo queries for index information, including:

- Whether or not the index is case insensitive.
- The number of fields that make up the key for the index.
- The field map for the index.
- The size of the index key.

5.2.40. GetLastInsertID

Returns the ID generated for an AUTO_INCREMENT column by the previous query.

Syntax:

```
procedure GetLastInsertID: Int64;
```

Description:

Use this function after you have performed an INSERT query into a table that contains an AUTO_INCREMENT field.

Note that **GetLastInsertID** returns **0** if the previous query does not generate an AUTO_INCREMENT value. If you need to save the value for later, be sure to call **GetLastInsertID** immediately after the query that generates the value.

Also note that the value of the SQL LAST_INSERT_ID function always contains the most recently generated AUTO_INCREMENT value, and is not reset between queries because the value of that function is maintained in the server.

5.2.41. GotoBookmark

Implements a virtual method to position the cursor on the record pointed to by a specified bookmark.

Syntax:

```
procedure GotoBookmark(Bookmark: TBookmark);
```

Description:

GotoBookmark calls a virtual, abstract internal method that is not implemented by **TDataSet**. Descendants of **TDataSet** redeclare and implement the internal method so that **GotoBookmark** makes the record identified by the **Bookmark** parameter active.

See also: Example: GetBookmark, GotoBookmark, FreeBookmark, FindPrior, Value, OnDataChange, BOF

5.2.42. Insert

Inserts a new, empty record in the dataset.

Syntax:

```
procedure Insert;
```

Description:

Call **Insert** to:

- Open a new, empty record in the dataset.
- Set the active record to the new record.

After a call to Insert, an application can enable users to enter data in the fields of the record, and then post those changes to the database using **Post** (or **ApplyUpdates** if cached updating is enabled).

For MySQL indexed tables, the index is updated with the new record information.

See also: Example: BeforeInsert, Insert, AsInteger, FieldByName

5.2.43. InsertRecord

Inserts a new, populated record to the dataset and posts it to the database.

Syntax:

```
procedure InsertRecord(const Values: array of const);
```

Description:

Call **InsertRecord** to create a new, empty record at in the dataset, populate it with the field values in Values, and post the values to the database.

For MySQL indexed tables, the index is updated with the new record information.

The newly inserted record becomes the active record.

Example:

This statement appends a record to the *Customer* table. Note that **Nulls** are entered for some of the values, but are not required for missing values at the end of the array argument.

```
Customer.InsertRecord([CustNoEdit.Text,
                       CoNameEdit.Text,
                       AddrEdit.Text,
                       Null,
                       Null,
                       Null,
                       Null,
                       Null,
                       Null,
                       DiscountEdit.Text]);
```

5.2.44. IsEmpty

Indicates whether the dataset contains no records.

Syntax:

```
function IsEmpty: Boolean;
```

Description:

Call **IsEmpty** to determine if a dataset has records. **IsEmpty** returns **True** if the dataset does not contain any records. Otherwise it returns **False**.

5.2.45. IsLinkedTo

Indicates whether a dataset is linked to a specified data source.

Syntax:

```
function IsLinkedTo(DataSource: TDataSource): Boolean;
```

Description:

Call **IsLinkedTo** to verify that a dataset is linked to a specific data source. **DataSource** is the name of the data source against which to test.

IsLinkedTo is mainly provided for developers deriving custom components based on **TDataSet**. It is not intended or needed for general data access.

If the datasource already provides data from the dataset or one of its nested dataset fields (or a nested dataset nested in a dataset field...), **IsLinkedTo** returns **True**. If the datasource provides data from some other dataset, or if the data source does not already have a dataset of its own, **IsLinkedTo** returns **False**.

5.2.46. Last

Positions the cursor on the last record in the dataset.

Syntax:

```
procedure Last;
```

Description:

Call **Last** to position the cursor on the last record in the dataset and make it the active record. **Last** posts any changes to the active record and:

- Clears the record buffers.
- Sets the cursor to the end of the dataset.
- Fetches the last record, positions the cursor on it, and makes it the active record.
- Fetches any additional records required for display, such as those needed to fill out a grid control.
- Sets the **Eof** property to **True**.
- Broadcasts the record change so that data controls and linked detail sets can update.

TDataSet uses internal, protected methods to position the database cursor and to fetch additional records required for display. In **TDataSet**, these internal methods are abstract. Descendant classes implement these methods to enable the Last method to work.

5.2.47. Locate

Searches the dataset for a specified record and makes that record the current record.

Syntax:

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

Description:

Call **Locate** to search a dataset for a specific record and position the cursor on it.

KeyFields is a string containing a semicolon-delimited list of field names on which to search.

KeyValues is a variant array containing the values to match in the key fields. If **KeyFields** lists a single field, **KeyValues** specifies the value for that field on the desired record. To specify multiple search values, pass a variant array as **KeyValues**, or construct a variant array on the fly using the **VarArrayOf** routine.

For example:

```
with CusTMySQLTable do
  Locate('Company;Contact;Phone',
         VarArrayOf(['Sight Diver',
         'P',
         '408-431-1000']), [loPartialKey]);
```

Options is a set that optionally specifies additional search latitude when searching on string fields. If **Options** contains the **loCaseInsensitive** setting, then **Locate** ignores case when matching fields.

If **Options** contains the **loPartialKey** setting, then **Locate** allows partial-string matching on strings in **KeyValues**.

If **Options** is an empty set, or if the **KeyFields** property does not include any string fields, **Options** is ignored.

Locate returns **True** if it finds a matching record, and makes that record the current one. Otherwise **Locate** returns **False**.

Locate uses the fastest possible method to locate matching records. If the search fields in **KeyFields** are indexed and the index is compatible with the specified search options, **Locate** uses the index. Otherwise **Locate** creates a filter for the search.

> **Delphi 7** and prior has poor support for *int64* values in *variant* type. This means that you'll be unable to use Locate and similar methods with such fields. Lookup fields and master-detail tables will not work properly too because of their dependence on **Locate** method. Please update your IDE to BDS 2006 (or later) or don't use BIGINT and UNSIGNED INT datatypes for keys and indexes if you need to use them in lookup fields. Also you can't use variant-style properties (**FieldValues**, **AsVariant** and so on) with such fields.

5.2.48. Lookup

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

Syntax:

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

Description:

Call **Lookup** to retrieve values for specified fields from a record that matches search criteria.

KeyFields
A string containing a semicolon-delimited list of field names on which to search.

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

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

Lookup returns a variant array containing the values from the fields specified in **ResultFields**. Otherwise it returns a Variant with the value Null, indicating that a matching record was not found.

Lookup uses the fastest possible method to locate matching records. If the search fields in **KeyFields** are indexed, **Lookup** uses the index. Otherwise **Lookup** creates a filter for the search.

Delphi 7 and prior has poor support for *int64* values in *variant* type. This means that you'll be unable to use Locate and similar methods with such fields. Lookup fields and master-detail tables will not work properly too because of their dependence on **Locate** method. Please update your IDE to BDS 2006 (or later) or don't use BIGINT and UNSIGNED INT datatypes for keys and indexes if you need to use them in lookup fields. Also you can't use variant-style properties (**FieldValues**, **AsVariant** and so on) with such fields.

See also: Locate method

5.2.49. MoveBy

Positions the cursor on a record relative to the active record in the dataset.

Syntax:

```
function MoveBy(Distance: Integer): Integer;
```

Description:

Call **MoveBy** to position the cursor on a record relative to the active record in the dataset. **Distance** indicates the number of records to move. A positive value for **Distance** indicates forward progress through the dataset, while a negative value indicates backward progress.

For example, the following statement moves backward through the dataset by **10** records:

```
MoveBy(-10);
```

MoveBy posts any changes to the active record and:

- Sets the **Bof** and **Eof** properties to **False**.
- If **Distance** is positive, repeatedly fetches subsequent records (if possible), decrementing **Distance** until it is zero, positions the cursor on the last record fetched, and makes it the active record. If an attempt is made to move past the end of the file, **MoveBy** sets **Eof** to **True**.
- If **Distance** is negative, repeatedly fetches previous records (if possible), incrementing **Distance** until it is zero, positions the cursor on the last record fetched, and makes it the active record. If an attempt is made to move past the start of the file, **MoveBy** sets **Bof** to **True**.
- Broadcasts information about the record change so that data-aware controls and linked datasets can update.
- Returns the actual number of records moved. In most cases, **Result** is the absolute value of **Distance**, but if **MoveBy** encounters the beginning-of-file or end-of-file before moving **Distance** records, **Result** will be less than the absolute value of **Distance**.

See also: Example: MoveBy, SelectedIndex, Tag

5.2.50. Next

Positions the cursor on the next record in the dataset.

Syntax:

```
procedure Next;
```

Description:

Call **Next** to position the cursor on the next record in the dataset and make it the active record. **Next** posts any changes to the active record and:

- Sets the **Bof** and **Eof** properties to **False**.
- Fetches the next record, positions the cursor on it, and makes it the active record.
- Fetches any additional records required for display, such as those needed to fill out a grid control.
- Sets the **Eof** property to **True** if the cursor was already on the last record in the dataset.
- Broadcasts the record change so that data controls and linked detail sets can update.

TDataSet uses internal, protected methods to position the database cursor and to fetch additional records required for display. In **TDataSet**, these internal methods are abstract. Descendant classes implement these methods to enable the **Next** method to work.

See also: Example: Min, Max, Position, RecordCount, First, Next

5.2.51. Open

Opens the dataset.

Syntax:

```
procedure Open;
```

Description:

Call **Open** to set the **Active** property for the dataset to **True**. When **Active** is **True**, data can be read from and written to the database.

Setting **Active** to **True**:

- Triggers the **BeforeOpen** event handler if one is defined for the dataset.
- Sets the dataset state to **dsBrowse**.
- Opens cursor into the dataset, if appropriate (only applies to **TMySQLDataSet** and its descendants, **TMySQLQuery** and **TMySQLTable**).

- Triggers the **AfterOpen** event handler if one is defined for the dataset.

If an error occurs during the dataset open, dataset state is set to **dsInactive**, and the cursor is closed.

5.2.52. OpenDatabase

Opens the database that contains the dataset.

Syntax:

```
function OpenDatabase: TMySQLDataBase;
```

Description:

Call **OpenDatabase** to connect to the database that contains the dataset. The **DatabaseName** property specifies the database to open.

5.2.53. Post

Writes a modified record to the database.

Syntax:

```
procedure Post; override;
```

Description:

Call **Post** to write a modified record to the database. Dataset methods that change the dataset state, such as **Edit**, **Insert**, or **Append**, or that move from one record to another, such as **First**, **Last**, **Next**, and **Prior** automatically call **Post**.

See also: Example: Append, FieldValues, Post

5.2.54. Prepare

Sends a query for optimization prior to execution.

Syntax:

```
procedure Prepare;
```

Description:

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

Delphi automatically prepares a query if it is executed without first being prepared. After execution, Delphi unprepares the query. When a query will be executed a number of times, an application should always explicitly prepare the query to avoid multiple and unnecessary prepares and unprepares.

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

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

Note: Prepared statements support is available since MySQL 4.1. If you are using a lower version the Prepare call will be skipped.

5.2.55. Prior

Positions the cursor on the previous record in the dataset.

Syntax:

```
procedure Prior;
```

Description:

Call **Prior** to position the cursor on the previous record in the dataset and make it the active record. Prior posts any changes to the active record and:

- Sets the **Bof** and **Eof** properties to **False**.
- Fetches the previous record, positions the cursor on it, and makes it the active record.
- Fetches any additional records required for display, such as those needed to fill out a grid control.
- Sets the **Bof** property to **True** if the cursor was already on the first record in the dataset.
- Broadcasts the record change so that data controls and linked detail sets can update.

TDataSet uses internal, protected methods to position the database cursor and to fetch additional records required for display. In **TDataSet**, these internal methods are abstract. Descendant classes implement these methods to enable the **Prior** method to work.

5.2.56. Refresh

Refetches data from the database to update a dataset's view of data.

Syntax:

```
procedure Refresh;
```

Description:

Call **Refresh** to ensure that an application has the latest data from a database. For example, when an application turns off filtering for a dataset, it should immediately call **Refresh** to display all records in the dataset, not just those that used to meet the filter condition.

See also: Example: SetRange, CancelRange, Refresh

5.2.57. RefreshRecord

Rereads the field values of the current record from a data source.

Syntax:

```
function RefreshRecord: Boolean;
```

Description:

Use **RefreshRecord** to discard all the changes of the current record and reread it from a data source.

To reread the record, DAC for MySQL performs the following sequence of steps:

- If **TMySQLDataSet.UpdateObject** is assigned and **TMySQLUpdateSQL.RefreshRecordSQL** is not empty, then this SQL command is executed.
- Otherwise DAC for MySQL generates a SELECT command that rereads a single row from the database and executes the command.

If the query to a data source returns no rows (for example, when a record is deleted), then when **TMySQLDataSet.Options.RefreshDelete** is True, a record is removed from a dataset, otherwise an exception is raised.

The method returns **True** if a record is refreshed. Otherwise, it returns **False** if it is deleted from a dataset.

5.2.58. Resync

Refetches the active record and the records that precede and follow it.

Syntax:

```
type TResyncMode = set of (rmExact, rmCenter);
procedure Resync(Mode: TResyncMode); virtual;
```

Description:

Resync is used internally by other dataset methods to resynchronize the Delphi dataset with underlying physical data when making calls that may change the cursor position. Applications should use the Refresh method instead of calling **Resync**.

Mode indicates optional processing that **Resync** should handle. If **Mode** contains **rmExact**, **Resync** raises an exception if **Resync** is called when the cursor is not positioned on a valid record. If **Mode** contains **rmCenter**, **Resync** positions the active record in the center of all buffered records.

Regardless of **Mode**, **Resync** also activates the buffers for the active record, retrieves prior and subsequent records for display purposes, and triggers a dataset change event.

5.2.59. RevertRecord

Restores the current record in the dataset to an unmodified state when cached updates are enabled.

Syntax:

```
procedure RevertRecord;
```

Description:

Call **RevertRecord** to undo changes made to the current record when cached updates are enabled.

To undo all changes to all pending updates in the cache, call **CancelUpdates**.

5.2.60. SetFields

Sets the values for all fields in a record.

Syntax:

```
procedure SetFields(const Values: array of const);
```

Description:

Call **SetFields** to set values for some or all fields in the active record at the same time.

Values contains the values to insert into each field. Values are assigned to the record based on the order of columns in the table or tables underlying the dataset. These values can be literals, variables, NULL, or **nil**. If Values contains fewer values than there are fields in the record, all records for which values are not provided are assigned a NULL value. A NULL value overwrites any existing value in such fields.

Before calling **SetFields**, call **Edit** to put the dataset into **dsEdit** state. After calling **SetFields**, call **Post** to write the changes to the database.

> To set values for some fields while retaining existing values for others, pass **nil** for each field that should not change.

5.2.61. SortBy

Sorts opened dataset on client side without refetching data from server.

Syntax:

```
procedure SortBy(FieldNames : string);
```

Description:

Call **SortBy** to sort dataset by particular fields on client side. Field names are case-sensitive and separated by commas. Every field name can be followed by the keyword *'ASC'* or *'DESC'* to specify a sort direction for the field. If one of these keywords is not used, the default sort direction for the field is ascending (*'ASC'*).

For example:

```
mySQLQuery1.SortBy('ID, Name DESC, ColorValue ASC');
```

Since v2.6.3 double-quote character (") can be used to quote field name if one contain spaces, commas or other non-alphanumeric character.

For example:

```
mySQLQuery1.SQL.Clear;
mySQLQuery1.SQL.Add('SELECT LEFT(TABLE_NAME, 20), TABLE_COLLATION FROM
INFORMATION_SCHEMA.TABLES');
mySQLQuery1.Open;
mySQLQuery1.SortBy('TABLE_COLLATION, "LEFT(TABLE_NAME, 20)" DESC');
```

This method affects dataset order only when it is opened, i.e. when **Active** = **True**.

Since v2.7.6 you can adjust case sensitivity of sorting using TMySQLDatabase.DatasetOptions property

Example:

This code can be used to sort data in **TDBGrid** component by particular column when user clicks on it title.

```
procedure TForm1.DBGrid1TitleClick(Column: TColumn);
begin
   mySQLTable1.SortBy(Column.FieldName + ' ASC');
end;
```

See also: SortFieldNames, TMySQLDatabase.DatasetOptions properties

5.2.62. Translate

Converts a data string between the ANSI character set used by Delphi (and Windows), and the local code page (OEM character set).

Syntax:

```
procedure Translate(Src, Dest: PChar; ToOem: Boolean); override;
```

Description:

When the **ToOem** parameter is **True**, **Translate** converts the source string from the ANSI character set to the OEM character set. If **ToOem** is **False**, **Translate** converts the source string from the OEM character set to the ANSI character set.

Before translation DAC for MySQL components compare code page on server with the client one automatically.

5.2.63. UnPrepare

Frees the resources allocated for a previously prepared query.

Syntax:

```
procedure UnPrepare;
```

Description:

Call **UnPrepare** to free the resources allocated for a previously prepared query on the server and client sides.

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

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

5.2.64. UpdateCursorPos

Positions the cursor on the active record.

Syntax:

```
procedure UpdateCursorPos;
```

Description:

UpdateCursorPos is an internal routine used by many dataset methods to ensure that the physical cursor is positioned on the active record. Normally an application should not need to call **UpdateCursorPos**. Typically **UpdateCursorPos** is called to ensure that the physical cursor position matches the logical cursor position.

5.2.65. UpdateRecord

Ensures that data-aware controls and detail datasets reflect record updates.

Syntax:

```
procedure UpdateRecord;
```

Description:

UpdateRecord is used internally by some dataset methods to inform data-aware controls of updates and trigger an **OnUpdateRecord** event if cached updates are enabled. Applications should not need to call **UpdateRecord** directly unless they provide custom dataset methods that bypass dataset methods.

5.2.66. UpdateStatus

Reports the update status for the current record.

Syntax:

```
function UpdateStatus: TUpdateStatus;
```

Description:

Call **UpdateStatus** to determine the update status for the current record in a dataset when cached updates are enabled. Update status can change frequently as records are edited, inserted, or deleted. **UpdateStatus** offers a convenient method for applications to assess the current status before undertaking or completing operations that depend on the update status of individual records in the dataset.

5.2.67. FetchNext

Retrieves the next block of records from a fetch on demand dataset.

Syntax:

```
procedure FetchNext;
```

Description:

Call **FetchNext** to retrieve the next block of records from your fetch on demand dataset. Count of records in this block depends from the TMySQLDataSet.Options.FetchRows property.

> For using then **FetchNext** method you need to enable the TMySQLDataSet.Options.FetchOnDemand property.

5.3. Events

Please see TMySQLDataset events short descriptions below:

Derived from TDataSet

AfterCancel
Occurs after an application completes a request to cancel modifications to the active record.

AfterClose
Occurs after an application closes a dataset.

AfterDelete
Occurs after an application deletes a record.

AfterEdit
Occurs after an application starts editing a record.

AfterInsert
Occurs after an application inserts a new record.

AfterOpen
Occurs after an application completes opening a dataset and before any data access occurs.

AfterPost
Occurs after an application writes the active record to the database or cache returns to browse state.

AfterRefresh
Occurs after an application refreshes the data in the dataset.

AfterScroll
Occurs after an application scrolls from one record to another.

BeforeCancel
Occurs before an application executes a request to cancel changes to the active record.

BeforeClose
Occurs before an application executes a request to close the dataset.

BeforeDelete
Occurs before an application attempts to delete the active record.

BeforeEdit
Occurs before an application enters edit mode for the active record.

BeforeInsert
Occurs before an application enters insert mode.

BeforeOpen
Occurs before an application executes a request to open a dataset.

BeforePost
Occurs before an application posts changes for the active record to the database or cache.

BeforeRefresh
Occurs immediately before an application refreshes the data in the dataset.

BeforeScroll
Occurs before an application scrolls from one record to another.

OnCalcFields
Occurs when an application recalculates calculated fields.

OnDeleteError
Occurs when an application attempts to delete a record and an exception is raised.

OnEditError
Occurs when an application attempts to modify or insert a record and an exception is raised.

OnFilterRecord
Occurs each time a different record in the dataset becomes the active record and filtering is enabled.

OnNewRecord
Occurs when an application inserts or appends a new dataset record.

OnPostError
Occurs when an application attempts to modify or insert a record and an exception is raised.

In TMySQLDataSet

OnUpdateError
Occurs if an exception is generated when cached updates are applied to a database.

OnDeleting
Occurs after all checks before data deleting are passed but before an application deletes this record from the database. This event allows to cancel record deleting.

OnInserting
Occurs after all checks before data insert are passed but before an application posts changes for this new record to the database. This event allows to cancel record insertion.

OnPosting
Occurs after all checks before post are passed but before an application posts changes for the active record to the database. This event allows to cancel data modification.

5.3.1. AfterCancel

Occurs after an application completes a request to cancel modifications to the active record.

Syntax:

```
property AfterCancel: TDataSetNotifyEvent;
```

Description:

Write an **AfterCancel** event handler to take specific action after an application cancels changes to the active record. **AfterCancel** is called by the **Cancel** method after it updates the cursor position, releases the lock on the active record if necessary, and sets the dataset state to **dsBrowse**. If an application requires additional processing before returning control to a user after a Cancel event, code it in the **AfterCancel** event.

Example:

This example updates the form's status bar with a message when an **AfterCancel** event occurs.

```
procedure TForm1.Table1AfterCancel(DataSet: TDataSet);
begin
  with TDataSet as TMySQLTable do
    StatusBar1.SimpleText := 'Record changes cancelled for ' + TableName;
end;
```

5.3.2. AfterClose

Occurs after an application closes a dataset.

Syntax:

```
property AfterClose: TDataSetNotifyEvent;
```

Description:

Write an **AfterClose** event handler to take specific action immediately after an application closes a dataset. For example, as a security measure, an application might clear a PASSWORD entry from the **Params** property of a database component when the dataset is closed.

AfterClose is called after a dataset is closed and the dataset state is set to **dsInactive**.

5.3.3. AfterDelete

Occurs after an application deletes a record.

Syntax:

```
property AfterDelete: TDataSetNotifyEvent;
```

Description:

Write an **AfterDelete** event handler to take specific action immediately after an application deletes the active record in a dataset. **AfterDelete** is called by **Delete** after it deletes the record, sets the dataset state to **dsBrowse**, and repositions the cursor on the record prior to the one just deleted.

See also: Example: AfterDelete, Format

5.3.4. AfterEdit

Occurs after an application starts editing a record.

Syntax:

```
property AfterEdit: TDataSetNotifyEvent;
```

Description:

Write an **AfterEdit** event handler to take specific action immediately after dataset enters edit mode. **AfterEdit** is called by **Edit** after it enables editing of a record, recalculates calculated fields, and calls the data event handler to process a record change.

Example:

This example updates the form's status bar with a message when an **AfterEdit** event occurs.

```
procedure TForm1.Table1AfterEdit(DataSet: TDataSet);
begin
   StatusBar1.SimpleText := 'Editing record';
end;
```

5.3.5. AfterInsert

Occurs after an application inserts a new record.

Syntax:

```
property AfterInsert: TDataSetNotifyEvent;
```

Description:

Write an **AfterInsert** event handler to take specific action immediately after an application inserts a record. The Insert and Append methods generate an **AfterInsert** event after inserting or appending a new record.

Example:

This example updates the form's status bar with a message when an **AfterInsert** event occurs.

```
procedure TForm1.MySQLTable1AfterInsert(DataSet: TDataSet);
begin
   StatusBar1.SimpleText := 'Inserting new record';
end;
```

5.3.6. AfterOpen

Occurs after an application completes opening a dataset and before any data access occurs.

Syntax:

```
property AfterOpen: TDataSetNotifyEvent;
```

Description:

Write an **AfterOpen** event handler to take specific action immediately after an application opens the dataset. **AfterOpen** is called after the cursor for the dataset is opened and the dataset is put into **dsBrowse** state. For example, an **AfterOpen** event handler might check the system registry to determine the last record touched in the dataset the previous time the application ran, and position the cursor at that record.

Example:

This example updates the form's status bar with a message when an **AfterOpen** event occurs.

```
procedure TForm1.Table1AfterOpen(DataSet: TDataSet);
begin
// now that the table is open, record information is available
 StatusBar1.SimpleText := 'Record ' +
                           IntToStr(MySQLTable1.RecNo) +
                          'from '   +
                           IntToStr(MySQLTable1.RecordCount);
end;
```

5.3.7. AfterPost

Occurs after an application writes the active record to the database or cache returns to browse state.

Syntax:

```
property AfterPost: TDataSetNotifyEvent;
```

Description:

Write an **AfterPost** event handler to take specific action immediately after an application posts a change to the active record. **AfterPost** is called after a modification, deletion, or insertion is made to a record.

Example:

This example updates the form's status bar with a message when an **AfterPost** event occurs.

```
procedure TForm1.MySQLTable1AfterPost(DataSet: TDataSet);
begin
  StatusBar1.SimpleText := 'Record changes complete';
end;
```

5.3.8. AfterRefresh

Occurs after an application refreshes the data in the dataset.

Syntax:

```
property AfterRefresh: TDataSetNotifyEvent;
```

Description:

Write an **AfterRefresh** event handler to take specific action immediately after an application has updated the records in the dataset. **AfterRefresh** is generated by calls to the **Refresh** method.

5.3.9. AfterScroll

Occurs after an application scrolls from one record to another.

Syntax:

```
property AfterScroll: TDataSetNotifyEvent;
```

Description:

Write an **AfterScroll** event handler to take specific action immediately after an application scrolls to another record as a result of a call to the **First**, **Last**, **MoveBy**, **Next**, **Prior**, **FindKey**, **FindFirst**, **FindNext**, **FindLast**, **FindPrior**, and **Locate** methods. **AfterScroll** is called after all other events triggered by these methods and any other methods that switch from record to record in the dataset.

5.3.10. BeforeCancel

Occurs before an application executes a request to cancel changes to the active record.

Syntax:

```
property BeforeCancel: TDataSetNotifyEvent;
```

Description:

Write a **BeforeCancel** event to take specific action before an application carries out a request to cancel changes. **BeforeCancel** is called by the Cancel method before it cancels a dataset operation such as **Edit**, **Insert**, or **Delete**.

An application might use the **BeforeCancel** event to record a user's changes in an undo buffer.

5.3.11. BeforeClose

Occurs before an application executes a request to close the dataset.

Syntax:

```
property BeforeClose: TDataSetNotifyEvent;
```

Description:

Write a **BeforeClose** event to take specific action before an application closes a dataset. Calling **Close** or setting the **Active** property to **False** results in a call to the **BeforeClose** event handler.

5.3.12. BeforeDelete

Occurs before an application attempts to delete the active record.

Syntax:

```
property BeforeDelete: TDataSetNotifyEvent;
```

Description:

Write a **BeforeDelete** event handler to take specific action before an application deletes the active record. **BeforeDelete** is called by **Delete** before it actually deletes a record.

Making use of this event an application might, for example, display a dialog box asking for confirmation before deleting the record. On denial of confirmation, the application could abort the deletion by calling the **Abort** procedure.

5.3.13. BeforeEdit

Occurs before an application enters edit mode for the active record.

Syntax:

```
property BeforeEdit: TDataSetNotifyEvent;
```

Description:

Write a **BeforeEdit** event handler to take specific action before an application enables editing of the active record.

For example, an application might keep a log of database edits, and therefore might record the edit request, time, and user in a **BeforeEdit** event before entering edit state.

5.3.14. BeforeInsert

Occurs before an application enters insert mode.

Syntax:

```
property BeforeInsert: TDataSetNotifyEvent;
```

Description:

Write a **BeforeInsert** event handler to take specific action before an application inserts or appends a new record. The Insert or **Append** method generates a **BeforeInsert** method before it sets the dataset into **dsInsert** state.

See also: Example: BeforeInsert, Insert, AsInteger, FieldByName

5.3.15. BeforeOpen

Occurs before an application executes a request to open a dataset.

Syntax:

```
property BeforeOpen: TDataSetNotifyEvent;
property read-write
```

Description:

Write a **BeforeOpen** event handler to take specific action before an application opens a dataset for viewing or editing. **BeforeOpen** is triggered when an application sets the **Active** property to **True** for a dataset or an application calls **Open**.

5.3.16. BeforePost

Occurs before an application posts changes for the active record to the database.

Syntax:

```
property BeforePost: TDataSetNotifyEvent;
```

Description:

Write a **BeforePost** event handler to take specific action before an application posts dataset changes to the database. **BeforePost** is triggered when an application calls the **Post** method. **Post** checks to make sure all required fields are present, then calls **BeforePost** before posting the record.

An application might use **BeforePost** to perform validity checks on data changes before posting them to the database. If it encountered a validity problem, it could call **Abort** to cancel the **Post** operation.

See also: Example: BeforePost,Abort

5.3.17. BeforeRefresh

Occurs immediately before an application refreshes the data in the dataset.

Syntax:

```
property BeforeRefresh: TDataSetNotifyEvent;
```

Description:

Write a **BeforeRefresh** event handler to take specific action immediately before an application updates the records in the dataset. **BeforeRefresh** is generated by calls to the **Refresh** method.

5.3.18. BeforeScroll

Occurs before an application scrolls from one record to another.

Syntax:

```
property BeforeScroll: TDataSetNotifyEvent;
```

Description:

Write a **BeforeScroll** event handler to take specific action immediately before an application scrolls to another record as a result of a call to the **First**, **Last**, **MoveBy**, **Next**, **Prior**, **FindKey**, **FindFirst**, **FindNext**, **FindLast**, **FindPrior**, and **Locate** methods. **BeforeScroll** is called before all other events triggered by these methods and any other methods that switch from record to record in the dataset.

5.3.19. OnCalcFields

Occurs when an application recalculates calculated fields.

Syntax:

```
property OnCalcFields: TDataSetNotifyEvent;
```

Description:

Write an **OnCalcFields** event handler to take specific action when an application recalculates calculated fields. A calculated field is one that derives its value from the values in one or more fields in the dataset, sometimes with additional processing.

When the **AutoCalcFields** property is **True**, **OnCalcFields** is triggered when:

- A dataset is opened.
- A dataset is put into **dsEdit** state.
- Focus moves from one visual control to another, or from one column to another is a data-aware grid control and modifications were made to the record.

- A record is retrieved from a database.

When the **AutoCalcFields** property is **True**, an **OnCalcFields** event handler should not modify the dataset (or a linked dataset if it is part of a master-detail relationship), because such modifications retrigger the **OnCalcField** event, leading to recursion.

If an application permits users to change data, **OnCalcFields** is frequently triggered. To reduce the frequency with which **OnCalcFields** occurs, set **AutoCalcFields** to **False**. When **AutoCalcFields** is **False**, **OnCalcFields** is not called when changes are made to individual fields within a record.

When the dataset is the master table of a master-detail relationship, **OnCalcFields** occurs before detail sets have been synchronized with the master table.

5.3.20. OnCompare

Occurs during the sorting fields.

Syntax:

```
property OnCompare: TCompareDataEvent;
```

Description:

Write a **OnCompare** event handler to perform custom client-side sortings.

See also: TDataSet.Methods.SortBy

5.3.21. OnDeleteError

Occurs when an application attempts to delete a record and an exception is raised.

Syntax:

```
type TDataSetErrorEvent = procedure(DataSet: TDataSet;
                                    E: EDatabaseError;
                                var Action: TDataAction) of object;
property OnDeleteError: TDataSetErrorEvent;
```

Description:

Write an **OnDeleteError** event handler to handle exceptions that occur when an attempt to delete a record fails.

DataSet is the dataset that failed to delete a record. **E** is a pointer to the database error object that contains the exception error message so that an application can display an error message. Action indicates how the dataset should respond to the error.

When **OnDeleteError** is first invoked, **Action** is always set to **daFail**. If the error handler can correct the error condition that caused the handler to be invoked, set **Action** to **daRetry** before exiting the handler. When **Action** is **daRetry**, the delete operation is tried again. If the error condition cannot be corrected, the display of the error message can be suppressed, if desired, by setting **Action** to **daAbort** instead of **daFail**.

5.3.22. OnDeleting

Since v2.6.0

Occurs after all checks before data deleting are passed but before an application deletes this record from the database. This event allows to cancel record deleting.

Syntax:

```
type
  TPostDataEvent = procedure(Sender: TObject; var Allow: boolean) of object;
property OnDeleting: TPostDataEvent;
```

Description:

Use this event if you need to check some custom conditions before delete active record from the database and disable this deleting in some cases. If you'll set **Allow** parameter value to **False** DAC for MySQL will not delete this record.

See also: OnPosting, OnInserting properties

5.3.23. OnEditError

Occurs when an application attempts to modify or insert a record and an exception is raised.

Syntax:

```
type TDataSetErrorEvent = procedure(DataSet: TDataSet;
                                    E: EDatabaseError;
                                var Action: TDataAction) of object;
property OnEditError: TDataSetErrorEvent;
```

Description:

Write an **OnEditError** event handler to handle exceptions that occur when an attempt to edit a record fails.

DataSet is the dataset that failed in editing a record. **E** is a pointer to the database error object that contains the exception error message so that an application can display an error message. **Action** indicates how the dataset should respond to the error.

When the **OnEditError** event handler is first invoked, **Action** is always set to **daFail**. If the error handler can correct the error condition that caused the handler to be invoked, set **Action** to **daRetry** before exiting the handler. When **Action** is **daRetry**, the edit operation is tried again. If an error condition cannot be corrected, the display of the error message can be suppressed, if desired, by setting **Action** to **daAbort** instead of **daFail**.

5.3.24. OnFilterRecord

Occurs each time a different record in the dataset becomes the active record and filtering is enabled.

Syntax:

```
type TFilterRecordEvent = procedure(DataSet: TDataSet;
                                    var Accept: Boolean) of object;
property OnFilterRecord: TFilterRecordEvent;
```

Description:

Write an **OnFilterRecord** event handler to specify for each record in a dataset whether it should be visible to the application. To indicate that a record passes the filter condition, an **OnFilterRecord** event handler must set the **Accept** parameter to **True**. To exclude a record, set the **Accept** parameter to **False**.

Filtering is enabled if the **Filtered** property is **True**. When an application is processing a filter, the **State** property for the dataset is **dsFilter**.

Use an **OnFilterRecord** event handler to filter records using a criterion that can't be implemented using the **Filter** property.

> Be sure that the interactions between the **Filter** property and the **OnFilterRecord** event handler do not result in an empty filter set when they are used simultaneously in an application.

Example:

The following example shows how to use a field comparison when filtering records on a local table, by using the **OnFilterRecord** event.

```
procedure TForm1.MySQLTable1FilterRecord(DataSet: TDataSet;
                                         var Accept: Boolean);
begin
  Accept := DataSet['DateOfPayment'] > DataSet['DateOfPurchase'] + 30;
end;
```

5.3.25. OnInserting

Since v2.6.0

Occurs after all checks before data insert are passed but before an application posts changes for this new record to the database. This event allows to cancel record insertion.

Syntax:

```
type
  TPostDataEvent = procedure(Sender: TObject; var Allow: boolean) of object;
property OnInserting: TPostDataEvent;
```

Description:

Use this event if you need to check some custom conditions before insert record to database and disable this insert in some cases. If you'll set **Allow** parameter value to **False** DAC for MySQL will not insert this record.

See also: OnPosting, OnDeleting properties

5.3.26. OnNewRecord

Occurs when an application inserts or appends a new dataset record.

Syntax:

```
property OnNewRecord: TDataSetNotifyEvent;
```

Description:

Write an **OnNewRecord** event handler to take specific actions as an application inserts or appends a new record. **OnNewRecord** is called as part of the actual insert or append process. An application might use the **OnNewRecord** event to set initial values for a record or as a way of implementing cascading insertions in related datasets.

5.3.27. OnPostError

Occurs when an application attempts to modify or insert a record and an exception is raised.

Syntax:

```
type TDataSetErrorEvent = procedure(DataSet: TDataSet;
                                    E: EDatabaseError;
                                var Action: TDataAction) of object;
property OnPostError: TDataSetErrorEvent;
```

Description:

Write an **OnPostError** event handler to handle exceptions that occur when an attempt to post a record fails.

DataSet is the dataset that failed to post a record. **E** is a pointer to the database error object that contains the exception error message so that an application can display an error message. **Action** indicates how the dataset should respond to the error.

When **OnPostError** is first invoked, **Action** is always set to **daFail**. If the error handler can correct the error condition that caused the handler to be invoked, set **Action** to **daRetry** before exiting the handler. When **Action** is **daRetry**, the post operation is tried again. If an error condition cannot be corrected, the display of the error message can be suppressed, if desired, by setting **Action** to **daAbort** instead of **daFail**.

5.3.28. OnPosting

Since v2.6.0

Occurs after all checks before post are passed but before an application posts changes for the active record to the database. This event allows to cancel data modification.

Syntax:

```
type
  TPostDataEvent = procedure(Sender: TObject; var Allow: boolean) of object;
property OnPosting: TPostDataEvent;
```

Description:

Use this event if you need to check some custom conditions before post data changes and disable post data changes. If you'll set **Allow** parameter value to **False** DAC for MySQL will not post data changes to server.

See also: OnInserting, OnDeleting properties

5.3.29. OnUpdateError

Occurs if an exception is generated when cached updates are applied to a database.

Syntax:

```
TUpdateErrorEvent = procedure(DataSet: TDataSet;
                              E: EDatabaseError;
                        UpdateKind: TUpdateKind;
                    var UpdateAction: TUpdateAction) of object;
property OnUpdateError: TUpdateErrorEvent;
```

Description:

Write an **OnUpdateError** event handler to respond to exceptions generated while applying cached updates to a database.

Because there is a delay between the time a record is first cached and the time cached updates are applied, there is a possibility that another application may change one or more of the same records in the database before the cached changes can be applied. MySQL checks for this condition and raises an exception. **TMySQLDataSet** responds by calling the **OnUpdateError** event handler if it exists.

DataSet is the name of the dataset to which updates are applied.

E is a pointer to a **EDatabaseError** object from which an application can extract an error message and the actual cause of the error condition. The **OnUpdateError** handler can use this information to determine how to respond to the error condition.

UpdateKind indicates whether the error occurred while inserting, deleting, or modifying a record.

UpdateAction indicates the action to take when the **OnUpdateError** handler exits. On entry into the handler, **UpdateAction** is always set to uaFail. If **OnUpdateError** can handle or correct the error, set **UpdateAction** to **uaRetry** before exiting the error handler.

The error handler can use the **TField.OldValue** and **TField.NewValue** properties to evaluate error conditions and set **TField.NewValue** to a new value to reapply. In this case, set **UpdateAction** to **uaRetry** before exiting.

If a call to **ApplyUpdates** raises an exception and **ApplyUpdates** is not called within the context of a *try...except* block, an error message is displayed. If an **OnUpdateError** handler cannot correct the error condition and leaves **UpdateAction** set to **uaFail**, the error message is displayed twice. To prevent redisplay, set **UpdateAction** to **uaAbort** in the error handler.

⚠ The code in an **OnUpdateError** handler must not call any methods that make a different record the current one.

6. TMySQLDirectQuery

Since v2.5.7

TMySQLDirectQuery component is intended for high-speed (3-4 times faster than with using of TMySQLQuery component) data fetching.

Description:

TMySQLDirectQuery allows to execute SQL queries and retrieve resultsets with very high performance. Meanwhile, it is not **TDataset**-compatible. This means that it can't be assigned to **TDatasource.Dataset** property and you can't use it with visual DB-controls. **TMySQLDirectQuery** is usually used in tasks where data require some processing without displaying it with/within visual DB-controls.

There are two nice examples of usage of this component:

1. Table2txt - This example shows how to save MySQL table data to a text file (one record per line). Fields values are delimited by TAB character. To increase performance we used **TMySQLQueryDirect** component.

2. TMySQLDirectQuery demo - allows you to compare **TMySQLDirectQuery** and **TMySQLQuery** performance on data fetching.

You are welcome to download examples at http://microolap.com/products/connectivity/mysqldac/download/

This component is used only for fetching data. That means that you can run queries that return resultset (SELECT, EXPLAIN, SHOW and so) with this component. If you need to run data modification or administration queries (INSERT, CREATE TABLE, COMMIT and so on) you can use TMySQLDatabase.Execute method. If you need to get single value from database (1 row X 1 field resultset) you can use one of TMySQLDatabase.SelectXXX methods.

See also: Properties, Methods

6.1. Properties

Please see TMySQLDirectQuery properties short descriptions below:

In TMySQLDirectQuery

Active
Specifies whether or not a dataset is open.

Bof
Indicates whether or not a cursor is positioned at the first record in a dataset.

Database
Specifies the **TMySQLDatabase** component this component connects to to perform database operations.

Eof
Indicates whether or not a cursor is positioned at the last record in a dataset.

FieldLength
Returns field databuffer size by field index.

FieldNames
Returns field name by it index.

FieldsCount
Indicates the number of fields in fetched resultset.

FieldTypes
Returns field type constant by it index.

FieldValues
Returns field value by it index.

IsEmpty
Indicates whether the dataset is empty.

RecNo
Indicates or sets the current record number in the dataset.

RecordCount
Indicates the total number of records associated with the dataset.

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

6.1.1. Active

Specifies whether or not a dataset is open.

Syntax:

```
property Active : boolean;
```

Description:

Use **Active** to determine or set a dataset's connection to data in a database. When **Active** is **False**, the dataset is closed; the dataset cannot read data from the database. When **Active** is **True**, data can be read from the database.

Unlike TMySQLQuery component Active property of TMySQLDirectQuery is not published. It is for run-time usage only.

An application must set **Active** to **False** before changing other properties that affect the status of the dataset.

> Calling the Open method sets **Active** to **True**; calling the Close method sets **Active** to **False**.

If an error occurs when setting **Active** property to **True** exception is raised and property value is set to **False**.

See also: Close, Open, Refresh methods

6.1.2. Bof

Indicates whether or not a cursor is positioned at the First record in a dataset.

Syntax:

```
property Bof : boolean;
```

Description:

Examine **Bof** (beginning-of-file) to determine if the cursor is positioned at the first record in a dataset. If **Bof** is **True**, the cursor is unequivocally on the first row in the dataset.

Bof is **True** when an application:

- Opens an empty dataset.
- Calls a dataset's First method.
- Call a dataset's Prior method, and the method fails (because the cursor is already on the first row in the dataset).
- **Bof** is **False** in all other cases.

If both Eof and **Bof** are **True**, the dataset is empty.

See also: IsEmpty property

6.1.3. Database

Specifies the TMySQLDatabase component this component connects to perform database operations.

Syntax:

```
property Database : TMySQLDatabase;
```

Description:

Use **Database** property to access the connection and some other properties, events, and methods of the database component associated with this dataset.

In design-time you may choose database from drop-down list for given TMySQLDirectQuery component.

See also: TMySQLDatabase component

6.1.4. Eof

Indicates whether or not a cursor is positioned at the last record in a dataset.

Syntax:

```
property Eof : boolean;
```

Description:

Examine **Eof** (end-of-file) to determine if the cursor is positioned at the last record in a dataset. If **Eof** is **True**, the cursor is unequivocally on the last row in the dataset.

Eof is **True** when an application:

- Opens an empty dataset.
- Calls a dataset's Last method.

- Call a dataset's Next method, and the method fails (because the cursor is already on the last row in the dataset).

Eof is **False** in all other cases.

> If both **Eof** and Bof are **True**, the dataset is empty.

See also: IsEmpty property

6.1.5. FieldLength

> Since v2.6.1

Returns field data buffer size by field index.

Syntax:

```
property FieldLength[aIndex : integer]: Cardinal;
```

Description:

Use **FieldLength** to get size of data buffer of field with index **aIndex** in active record. This can be useful if you want to process BLOB fields with 0x00 character within the data.

Use FieldRawDataPointer method to get a direct pointer to field data buffer.

See also: FieldRawDataPointer method

6.1.6. FieldNames

Returns field name by its index.

Syntax:

```
property FieldNames[aIndex : integer]: string;
```

Description:

Use **FieldNames** to get name of field with index **aIndex** in fetched resultset.

See also: FieldsCount property, FieldIndexByName method

6.1.7. FieldsCount

Indicates the number of fields in fetched resultset.

Syntax:

```
property FieldsCount : integer;
```

Description:

Examine **FieldsCount** property to determine the number of fields in fetched resultset.

See also: FieldNames property

6.1.8. FieldValues

Returns field value by it index.

Syntax:

```
property FieldValues[aIndex : integer]: string;
```

Description:

Use **FieldValues** to get value of field with index **aIndex** in active record.

- All fields values are returned as strings when using TMySQLDirectQuery component for performance reasons.
- NULL fields values are returned as empty strings. Use FieldIsNull method to distinguish empty string values from NULL values.

Use FieldRawDataPointer method if you need to work with BLOB fields.

Please read this FAQ section if you want to use Unicode strings in your application: How to use Unicode data in my application?

See also: FieldsCount property, FieldIsNull, FieldValueByName, FieldRawDataPointer methods

6.1.9. IsEmpty

Indicates whether the dataset is empty.

Syntax:

```
property IsEmpty : boolean;
```

Description:

Examine **IsEmpty** to determine if dataset has no records. **IsEmpty** is **True** if dataset contains **0** records.

See also: RecordCount, Bof, Eof properties

6.1.10. RecNo

Indicates or sets the current record number in the dataset.

Syntax:

```
property RecNo : int64;
```

Description:

Examine **RecNo** to determine the record number of the current record in the dataset. Applications might use this property with RecordCount to iterate through all the records in a dataset, though typically record iteration is handled with calls to First, Last, MoveBy, Next and Prior methods.

If accessing tables, **RecNo** can be set to a specific record number to position the cursor on that record, beginning from **0**.

See also: RecordCount property, First, Last, Next, Prior, MoveBy methods

6.1.11. RecordCount

Indicates the total number of records associated with the dataset.

Syntax:

```
property RecordCount : int64;
```

Description:

Examine **RecordCount** to determine the total number of records in the dataset. Applications might use this property with **RecNo** property to iterate through all the records in a dataset, though typically record iteration is handled with calls to First, Last, MoveBy, Next and Prior methods.

See also: RecNo property, First, Last, Next, Prior, MoveBy methods

6.1.12. SQL

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

Syntax:

```
property SQL : TStrings;
```

Description:

Use **SQL** to provide the SQL statement that a component executes when its Open method is called or Active property is set to **True**. At design time the SQL property can be edited by invoking the **String List editor** in the **Object Inspector**.

The **SQL** property may contain only one complete SQL statement at a time. In general, multiple "batch" statements are not allowed unless a particular server supports them.

Use TMySQLBatchExecute component if you want execute several queries in "batch" mode.

See also: Active property, Open method, TMySQLBatchExecute component

6.1.13. TMySQLDirectQuery.Properties.FieldTypes

Returns field type constant by it index.

Syntax:

```
property FieldTypes[aIndex : integer]: byte;
```

Description:

Use **FieldTypes** to get type of field constant defined in DAC for MySQL with index **aIndex** in fetched resultset.

See also: FieldsCount property, FieldIndexByName method

6.2. Methods

Please see TMySQLDirectQuery methods short descriptions below:

In TMySQLDirectQuery

Close
Closes the dataset.

FieldIndexByName
Returns field index by its name.

FieldIsNull
Used to determine if field value is NULL.

FieldRawDataPointer
Returns direct pointer to field databuffer by field index.

FieldValueByFieldName
Returns field value by it name.

First
Positions the cursor on the first record in the dataset.

Last
Positions the cursor on the last record in the dataset.

MoveBy
Positions the cursor on a record relative to the active record in the dataset.

Next
Positions the cursor on the last record in the dataset.

Open
Opens the dataset.

Prior
Positions the cursor on the previous record in the dataset.

Refresh
Refreshes the dataset.

6.2.1. Close

Closes the dataset.

Syntax:

```
procedure Close;
```

Description:

Call this method to close result set and to free resources. Active property is set to **False**.

> You can set Active property to **False** instead of calling this method.

See also: Active property, Open method

6.2.2. FieldIndexByName

Returns field index by its name.

Syntax:

```
function FieldIndexByName(aFieldName : string) : integer;
```

Description:

Function **FieldIndexByName** returns zero-based index of the field with name **aFieldName**.

See also: FieldsCount, FieldNames properties

6.2.3. FieldIsNull

Used to determine if field value is NULL.

Syntax:

```
function FieldIsNull(aFieldIndex : integer) : boolean;overload;
function FieldIsNull(aFieldName : string) : boolean;overload;
```

Description:

FieldIsNull are used to determine if field value is NULL in active record. First implementation checks field by it index. Second implementation checks field by it name.

Returns:

FieldIsNull returns **True** if value of requested field is NULL in current record and **False** if value is not NULL.

See also: FieldValues property, FieldValueByFieldName method

6.2.4. FieldRawDataPointer

Since v2.6.1

Returns direct pointer to field data buffer by the field index.

Syntax:

```
function FieldRawDataPointer(aFieldIndex : integer) : pointer;
```

Description:

Call **FieldRawDataPointer** method to get direct pointer to data buffer of field with **aFieldIndex** index in active record. This can be useful if you want to process BLOB fields with **0x00** character within it data.

Use FieldLength property to get field data buffer size.

See also: FieldLength property, FieldIndexByName method

6.2.5. FieldValueByFieldName

Returns field value by it name.

Syntax:

```
function FieldValueByFieldName(aFieldName : string) : string;
```

Description:

Call **FieldValueByFieldName** to get value of field with name **aFieldName** in active record.

- For performance reasons all fields values are returned as strings when using TMySQLDirectQuery component.
- NULL fields values are returned as empty string. Use FieldIsNull method to distinguish empty string values from NULL values.

Use FieldRawDataPointer method to work with BLOB fields.

Please read this FAQ section if you want to use Unicode strings in your application: How to use Unicode data in my application?

See also: FieldValues, FieldNames properties, FieldIsNull, FieldRawDataPointer methods

6.2.6. First

Positions the cursor on the first record in the dataset.

Syntax:

```
procedure First;
```

Description:

Call **First** to position the cursor on the first record in the dataset and make it the active record.

See also: RecNo property, Last, Next, Prior, MoveBy methods

6.2.7. Last

Positions the cursor on the last record in the dataset.

Syntax:

```
procedure Last;
```

Description:

Call **Last** to position the cursor on the last record in the dataset and make it the active record.

See also: RecNo property, First, Next, Prior, MoveBy methods

6.2.8. MoveBy

Positions the cursor on a record relative to the active record in the dataset.

Syntax:

```
function MoveBy(aDistance : int64) : int64;
```

Description:

Call **MoveBy** to position the cursor on a record relative to the active record in the dataset. **aDistance** indicates the number of records to move. A positive value for **aDistance** indicates forward progress through the dataset, while a negative value indicates backward progress.

For example, the following statement moves backward through the dataset by 10 records:

```
MoveBy(-10);
```

Function returns the actual number of records moved. In most cases, **Result** is the absolute value of **aDistance**, but if **MoveBy** encounters the beginning-of-file or end-of-file before moving **aDistance** records, Result will be less than the absolute value of **aDistance**.

See also: RecNo, RecordCount properties, First, Last, Next, Prior methods

6.2.9. Next

Positions the cursor on the last record in the dataset.

Syntax:

```
procedure Next;
```

Description:

Call **Next** to position the cursor on the next record in the dataset and make it the active record.

See also: RecNo property, First, Last, Prior, MoveBy methods

6.2.10. Open

Opens the dataset.

Syntax:

```
procedure Open;
```

Description:

Call this method to execute query provided with SQL property and to fetch resultset. Active property is set to **True** and you can read data from resultset after that.

> You can set Active property to **True** instead of calling this method.

If an error occurs during the dataset open exception is raised and Active property is set to **False**. Use Close method to close resultset and free resources.

See also: Active property, Close, Refresh methods

6.2.11. Prior

Positions the cursor on the previous record in the dataset.

Syntax:

```
procedure Prior;
```

Description:

Call **Prior** to position the cursor on the previous record in the dataset and make it the active record.

See also: RecNo property, First, Last, Next, MoveBy methods

6.2.12. Refresh

Refreshes the dataset.

Syntax:

```
procedure Refresh;
```

Description:

Call this method to close result set and run the same query again. This is the same as calling Close and Open methods one by one.

See also: Close, Open methods

7. TMySQLDump

 **TMySQLDump** allows to get SQL script with a dump of a Database. This script can be executed on another MySQL server by TMySQLBatchExecute component.

See also: Properties, Methods, Events

7.1. Properties

Please see TMySQLDump properties short descriptions below:

CompleteInsert
An option to include the field names in "insert" command

Database
Points to TMySQLDatabase component which sets a DB to be connected with.

Delimiter
Sets the SQL statements delimiter.

DisableKeys
Generates "ALTER TABLE <TableName> DISABLE KEYS" SQL statements.

DisableUniqueChecks
Generates SQL statements to disable uniqueness checks for secondary indexes in InnoDB tables.

DropObject
Generates DROP TABLE SQL statement.

DumpOption
Sets DB dump type.

ExcludeTables
Exclude the contents of specified tables from dump process.

ExtInsert
Generates INSERT SQL Statement using extended syntax.

ExtInsertsCount
Sets the number of <FIELDVALUES> sections for single INSERT statement with extended syntax.

IgnoreLockTables
Exclude the 'LOCK TABLE' clause of specified tables from dump process.

IncludeHeader
Sets if the dump information will be written at the beginning of the SQL file.

Limit
Sets maximum number of rows to be retrieved from server at once during dumping tables.

LineComment
Sets the line comment string for SQL statements.

LockTables
Specifies whether to add "LOCK TABLE" clause to the dump script.

RewriteFile
Sets SQL file opening mode.

SQLFile
Sets a full path to a file with SQL script will be generated.

TableList
Sets a list of DB tables to be dumped.

UseCreateDB
Generates CREATE DATABASE SQL statements.

UseHexBlob
Specifies whether to use hexadecimal notation when dumping binary columns.

7.1.1. CompleteInsert

Since v2.6.1

An option to include the field names in INSERT command.

Syntax:

```
CompleteInsert : Boolean;
```

Description:

CompleteInsert set in **True** generates INSERT SQL Statement with field names.

For example, **CompleteInsert = False** generates the following SQL statements:

```
INSERT INTO TableName VALUES (value1, value2, ...);
```

And **CompleteInsert = True** generates the following SQL statements:

```
INSERT INTO TableName(fieldname1, fieldname2, ...) VALUES (value1, value2, ...);
```

See also: ExtInsert property

7.1.2. Database

Points to TMySQLDatabase component which sets a DB to be connected with.

Syntax:

```
Database: TMySQLDatabase;
```

7.1.3. Delimiter

Sets the SQL statements delimiter.

Syntax:

```
Delimiter: Char;
```

Description:

SQL statements included into SQL script will be separated by the symbol defined in **Delimiter** property (";" by default).

See also: LineComment property

7.1.4. DisableKeys

Generates "ALTER TABLE <TableName> DISABLE KEYS" SQL statements.

Syntax:

```
property DisableKeys: Boolean default False;
```

Description:

Set **DisableKeys** property value to **True** to generate the following SQL statements.

In dump script header:

```
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0
*/;
```

In dump script footer:

```
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
```

Before table dump:

```
/*!40000 ALTER TABLE <TableName> DISABLE KEYS */;
```

After table dump:

```
/*!40000 ALTER TABLE <TableName> ENABLE KEYS */;
```

Disabling foreign key checking can be useful for reloading InnoDB tables in an order different from that required by their parent/child relationships. Take a look at MySQL Manual for details.

7.1.5. DisableUniqueChecks

Since v2.7.0

Generates SQL statements to disable uniqueness checks for secondary indexes in InnoDB tables.

Syntax:

```
property DisableUniqueChecks: Boolean default True;
```

Description:

Set **DisableUniqueChecks** property value to **True** to generate the following SQL statements.

In dump script header:

```
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
```

In dump script footer:

```
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
```

Take a look at MySQL Manual for additional details.

7.1.6. DropObject

Generates DROP TABLE SQL statement.

Syntax:

```
DropObject : Boolean;
```

Description:

DropObject set in **True** generates SQL statement:

```
DROP TABLE IF EXISTS <ObjectName>;
```

7.1.7. DumpOption

Sets DB dump type.

Syntax:

```
DumpOption : TDumpOption;
Type
TDumpOption = (dStructure,dData,dAll);
```

Description:

Allows to choose DB dump type:

dStructure
Generate SQL script containing the DB structure only;

dData
Generate SQL script containing the data only;

dAll
Generate SQL script containing both structure and data.

7.1.8. ExcludeTables

Exclude the contents of specified tables from dump process.

Syntax:

```
property ExcludeTables: TStrings;
```

Description:

Do not dump any tables matching the table pattern. The pattern is interpreted according to the same rules as for **TableList** property.

When both **TableList** and **ExcludeTables** are given, the behavior is to dump just the tables that match at least one **TableList** string but no **ExcludeTables** strings. If **ExcludeTables** appears without **TableList**, then tables matching **ExcludeTables** are excluded from what is otherwise a normal dump.

See also: TableList property

7.1.9. ExtInsert

Generates INSERT SQL Statement using extended syntax.

Syntax:

```
ExtInsert : Boolean;
```

Description:

ExtInsert set in **True** generates INSERT SQL Statement using extended syntax. For example, **ExtInsert = True** generates the series of SQL statements:

```
INSERT INTO (<FieldList>) VALUES (<FIELDVALUES>),(<FIELDVALUES>),...,
(<FIELDVALUES>);
```

Every single INSERT statement will contain ExtInsertsCount <FIELDVALUES> sections.

See also: ExtInsertsCount, CompleteInsert properties

7.1.10. ExtInsertsCount

Sets the number of <FIELDVALUES> sections for single INSERT statement with extended syntax.

Syntax:

```
property ExtInsertsCount : integer;
```

Description:

Every single INSERT statement will contain **ExtInsertsCount** <FIELDVALUES> sections.

```
INSERT INTO (<FieldList>) VALUES (<FIELDVALUES>),(<FIELDVALUES>),...,
(<FIELDVALUES>);
INSERT INTO (<FieldList>) VALUES (<FIELDVALUES>),(<FIELDVALUES>),...,
(<FIELDVALUES>);
...
INSERT INTO (<FieldList>) VALUES (<FIELDVALUES>),(<FIELDVALUES>),...,
(<FIELDVALUES>);
```

See also: ExtInsert property

7.1.11. IgnoreLockTables

Exclude the 'LOCK TABLE' clause of specified tables from dump process.

Syntax:

```
IgnoreLockTables : TStrings;
```

Description:

If there is a need to add "LOCK TABLE" clause not for each table, set property LockTables to **True** and specify tables for which there are no need to add this clause in **IgnoreLockTables** property.

7.1.12. IncludeHeader

Sets if the dump information will be written at the beginning of the SQL file.

Syntax:

```
property IncludeHeader : Boolean default True;
```

Description:

If **IncludeHeader** property is set in **True**, several comments strings with the information about dump version, host name, and database name will be included in the header of the SQL file defined in the SQLFile property.

7.1.13. Limit

Since v2.6.1

Sets maximum number of rows to be retrieved from server at once during dumping tables.

Syntax:

```
property Limit : Cardinal; default 1024;
```

Description:

This property value limits number of rows to be retrieved from server at once during dump process. For example, a table with 2050 rows will be retrieved in 3 times - the first 1024 rows, the second 1024, and the final 2 rows. This allows to limit memory usage for large tables dumping. You can reduce this value if you're dumping really huge BLOB fields.

This property doesn't correlate with ExtInsertCount property. ExtInsertCount property value is used when dumped data are being saved to a file, meanwhile Limit property value is used when data are being retrieved from server.

7.1.14. LineComment

Since v2.5.5

Sets the line comment string for SQL statements.

Syntax:

```
property LineComment  : string;
```

Description:

Comments included into SQL script will be preceded by the string defined in **LineComment** property ("--" by default).

7.1.15. LockTables

Specifies whether to add "LOCK TABLE" clause to the dump script.

Syntax:

```
LockTables : Boolean;
```

Description:

If **LockTables** set to **True** then for each dumped table in the script will be added "LOCK TABLE" clause.

See also: IgnoreLockTables property

7.1.16. RewriteFile

Sets SQL file opening mode.

Syntax:

```
property RewriteFile : Boolean default True;
```

Description:

If **RewriteFile** property is set in **True**, and file defined in the SQLFile property exists, then this file will be overwritten. Otherwise the dump will be appended to the end of file.

7.1.17. SQLFile

Sets a full path to a file with SQL script will be generated.

Syntax:

```
SQLFile : TFileName;
```

Description:

If **SQLFile** was not set, the current directory will be used and *<database name>.sql* file will be created.

7.1.18. TableList

Sets a list of DB tables to be dumped.

Syntax:

```
TableList : TSTrings;
```

Description:

Use **TableList** to set a list of DB tables to be dumped. If **TableList** is empty SQL script will not be generated.

7.1.19. UseCreateDB

Generates CREATE DATABASE SQL statements.

Syntax:

```
UseCreateDB : Boolean;
```

Description:

UseCreateDB set in **True** generates the following pair of SQL statements:

```
CREATE DATABASE /*!32312 IF NOT EXISTS*/ <DBName>;
USE <DBName>;
```

7.1.20. UseHexBlob

Dump binary columns using hexadecimal notation.

Syntax:

```
UseHexBlob : Boolean;
```

Description:

If **UseHexBlob** set to **True** then dump binary columns using hexadecimal notation. For example, *'abc'* becomes *0x616263*.

7.2. Methods

Please see TMySQLDump methods short descriptions below:

Execute
Generates SQL script.

DumpToStream
Generates SQL script to **TStream** descendant.

7.2.1. DumpToStream

Since v2.5.5

Generates SQL script to **TStream** descendant.

Syntax:

```
procedure DumpToStream(aStream: TStream);
```

Description:

DumpToStream method generates SQL script to **TStream** descendant (for example **TMemoryStream**, **TFileStream** and so on).

See also: Execute method

7.2.2. Execute

Generates SQL script to file.

Syntax:

```
function Execute: Boolean;
```

Description:

Execute function generates SQL script to file specified in **SQLFile** property. Execute returns **False** on error, and **True** on success.

See also: DumpToSream method, SQLFile property

7.3. Events

Please see TMySQLDump events short descriptions below:

BeforeDump
Occurs immediately before starting a dump process.

OnDataProcess
Occurs when each table data from TableList processing begins.

OnProcess
Occurs when each table from TableList processing begins.

7.3.1. BeforeDump

Occurs immediately before starting a dump process.

Syntax:

```
property BeforeDump: TNotifyEvent;
```

Description:

Write a **BeforeDump** event handler to take application-specific actions before the dump component starting a dump process.

See also: TMySQLDump.Methods.DumpToStream

7.3.2. OnDataProcess

Occurs when each table data from TableList processing begins.

Syntax:

```
type TOnDataProcess = procedure (Sender : TObject;
                                 Percent : Integer) of object;
OnDataProcess: TOnDataProcess;
```

Description:

Create **OnDataProcess** event handler to get information about current table SQL script generation process status.

Percent contains progress data in percents about current table SQL script generation process. Is valid if DumpOption property is set to **dData** or **dAll**.

7.3.3. OnProcess

Occurs when each table from TableList processing begins.

Syntax:

```
type TOnProcess = procedure (Sender : TObject;
                             Table : String;
                           Percent  : Integer) of object;
OnProcess : TOnProcess;
```

Description:

Create **OnProcess** event handler to get information about whole SQL script generation process status.

Table
Contains the name of the table which is currently in processing;

Percent
Contains progress data in percents about whole SQL script generation process. Is valid if DumpOption property is set to **dAll**.

8. TMySQLMacroQuery

 **TMySQLMacroQuery** is the descendant of TMySQLQuery component and supports all of its properties, methods, events, and functionalities. The difference is in Macros and MacroChar properties which help to modify SQL script text in design-time and run-time with easy.

See also: Properties, Methods, Events

8.1. Properties

Please see TMySQLMacroQuery properties short descriptions below:

Derived from TDataSet

Active
Specifies whether or not a dataset is open.

AutoCalcFields

Determines when the **OnCalcFields** event is triggered.

Bof
Indicates whether or not a cursor is positioned at the first record in a dataset.

Bookmark
Specifies the current bookmark in the dataset.

CachedUpdates
Does not affect on dataset behavior.

DefaultFields
Indicates whether a dataset's underlying field components are generated dynamically when the dataset is opened.

Eof
Indicates whether or not a cursor is positioned at the last record in a dataset.

FieldCount
Indicates the number of field components associated with the dataset.

FieldDefList
Points to the list of field definitions for the dataset.

FieldList
Lists the field components of a dataset.

Fields
Lists all non-aggregate field components of the dataset.

FieldValues
Provides access to the values for all fields in the active record for the dataset.

Found
Indicates whether or not moving to a different record is successful.

Modified
Indicates whether the active record is modified.

Name
Designates the name of the dataset as referenced by other components.

ObjectView
Specifies whether fields are to be stored hierarchically or flattened out in the Fields property.

SparseArrays
Determines whether a unique **TField** object is created for each element of an array field.

State
Indicates the current operating mode of the dataset.

Derived from TMySQLDataSet

AllowSequenced
Determines that database records can be located by sequence numbers.

AutoRefresh
Specifies whether server-generated field values are refetched automatically.

AvailableResultsetCount
Indicates count of resultsets available to fetch from multiresultset query or stored procedure. This property is useful when query or stored procedure returns more than one dataset.

BlockReadSize
Determines how many record buffers are read in each block.

CacheBlobs
Determines whether BLOB fields are cached in memory.

Database
Specifies the database component for which this dataset represents one or more tables.

Filter
Specifies the text of the current filter for a dataset.

Filtered
Specifies whether filtering is active for a dataset.

FilterOptions
Specifies whether filtering is case insensitive, and whether or not partial comparisons are permitted when filtering records.

KeySize
Specifies the size of the key for the current index of the dataset.

LastInsertID
Get last inserted value of AUTO_INCREMENT column from MySQL server.

RecNo
Indicates the current record in the dataset.

RecordCount
Indicates the total number of records associated with the dataset.

RecordSize
Indicates the size of a record in the dataset.

SortFieldNames
Specifies field names and sorting order to sort opened dataset by these fields on the client side without refetching data from server.

UpdateMode
Determines how MySQL finds records when updating to an SQL database.

UpdateObject
Specifies the update object component used to update a read-only result set.

In TMySQLQuery

DataSource
Specifies the data source component from which to extract current field values to use with same-name parameters in the query SQL statement.

Handle
Specifies the cursor handle for the query.

MultiResultsetNo
Specifies the resultset to associate with component when it become active. This property is useful when query or stored procedure returns more than one dataset.

ParamCheck
Specifies whether the parameter list for a query is regenerated if the SQL property changes at runtime.

ParamCount
Indicates the current number of parameters for the query.

Params
Contains the parameters for a query SQL statement.

Prepared
Determines whether or not a query is prepared for execution.

ProcessComments
Allows to choose what kind of comments to cut from SQL query text before sending it to server.

RequestLive
Specifies whether an application expects to receive a live result set when the query executes.

RowsAffected
Returns the number of rows operated upon by the latest query execution.

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

SQLBinary
Points to the binary data stream that represents an SQL query statement or result set.

Text
Points to the actual text of the SQL query passed to MySQL.

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

In TMySQLMacroQuery

MacroChar
Sets the macro definition symbol.

MacroCount
Contains the number of macro definitions in SQL query.

Macros
Contains Macros array with current query macro definitions.

8.1.1. MacroChar

Sets the macro definition symbol.

Syntax:

```
MacroChar: Char;
```

Description:

Sets a symbol which will be considered as the beginning of the macro in SQL script text. This symbol will be used the same way as symbol ":" intended for query parameters identification. The default value of MacroChar is "%".

8.1.2. MacroCount

Contains the number of macro definitions in SQL query.

Syntax:

```
MacroCount : Word;
```

Run-Time read-only property.

8.1.3. Macros

Contains Macros array with current query macro definitions.

Syntax:

```
Macros: TParams;
```

Description:

When you run SQL query with macro definitions included, **TMySQLMacroQuery** creates Macros array containing current query macro definitions. Use MacroCount property to get the number of macro definitions.

8.2. Methods

Please see TMySQLMacroQuery methods short descriptions below:

Derived from TDataSet

ActiveBuffer
Returns a pointer to the buffer for the active record.

Append
Adds a new, empty record to the end of the dataset.

AppendRecord
Adds a new, populated record to the end of the dataset and posts it to the database.

CheckBrowseMode
Automatically posts or cancels data changes when an application changes which record in the dataset is the active record.

ClearFields
Clears the contents of all fields for the active record.

Close
Closes a dataset.

ControlsDisabled
Indicates whether data-aware controls do not update their display to reflect changes to the dataset.

CursorPosChanged
Marks the internal cursor position as invalid.

Delete
Deletes the active record and positions the cursor on the next record.

DisableControls
Disables data display in data-aware controls associated with the dataset.

Edit
Enables editing of data in the dataset.

EnableControls
Re-enables data display in data-aware controls associated with the dataset.

FieldByName
Finds a field based on its name.

FindField
Searches for a specified field in the dataset.

FindFirst
Implements a virtual method for positioning the cursor on the first record in a filtered dataset.

FindLast
Implements a virtual method for positioning the cursor on the last record in a filtered dataset.

FindNext
Implements a virtual method for positioning the cursor on the next record in a filtered dataset.

FindPrior
Implements a virtual method for positioning the cursor on the previous record in a filtered dataset.

First
Positions the cursor on the first record in the dataset.

FreeBookmark
Frees the resources allocated for a specified bookmark.

GetBookmark
Allocates a bookmark for the current cursor position in the dataset.

GetDetailDataSets
Fills a list with a dataset for every detail dataset that is not the value of a nested dataset field.

GetFieldList
Retrieves a specified set of field objects into a list.

GetFieldNames
Retrieves a list of names for all fields in a dataset.

GotoBookmark
Implements a virtual method to position the cursor on the record pointed to by a specified bookmark.

Insert
Inserts a new, empty record in the dataset.

InsertRecord
Inserts a new, populated record to the dataset and posts it to the database.

IsEmpty
Indicates whether the dataset contains no records.

IsLinkedTo
Indicates whether a dataset is linked to a specified data source.

Last
Positions the cursor on the last record in the dataset.

MoveBy

Positions the cursor on a record relative to the active record in the dataset.

Next
Positions the cursor on the next record in the dataset.

Open
Opens the dataset.

Prior
Positions the cursor on the previous record in the dataset.

Refresh
Refetches data from the database to update a dataset's view of data.

Resync
Refetches the active record and the records that precede and follow it.

SetFields
Sets the values for all fields in a record.

UpdateCursorPos
Positions the cursor on the active record.

UpdateRecord
Ensures that data-aware controls and detail datasets reflect record updates.

Derived from TMySQLDataSet

ApplyUpdates
Writes a dataset's pending cached updates to the database.

BookmarkValid
Tests the validity of a specified bookmark.

Cancel
Cancels modifications to the current record if those changes are not yet posted.

CancelUpdates
Clears all pending cached updates from the cache and restores the dataset its prior state.

CheckOpen
Checks the result of a call to the MySQL.

CloseDatabase
Closes a database connection associated with the database.

CommitUpdates
Clears the cached updates buffer.

CompareBookmarks
Indicates the relationship between two bookmarks.

FetchAll

Retrieves all records from the current cursor position to the end of the file and stores them locally.

FlushBuffers
Posts all changes that have been written to the record buffer.

GetBlobFieldData
Reads BLOB data into a buffer.

GetCurrentRecord
Retrieves the current record into a buffer.

GetFieldData
Retrieves the current value of a field into a buffer.

GetIndexInfo
Retrieves information about the current index into the index data fields of the dataset.

GetLastInsertID
Returns the ID generated for an AUTO_INCREMENT column by the previous query.

Locate
Searches the dataset for a specified record and makes that record the current record.

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

OpenDatabase
Opens the database that contains the dataset.

Post
Writes a modified record to the database.

RevertRecord
Restores the current record in the dataset to an unmodified state when cached updates are enabled.

SortBy
Sorts opened dataset on client side without refetching data from server.

Translate
Converts a data string between the ANSI character set used by Delphi (and Windows), and the local code page (OEM character set).

UpdateStatus
Reports the update status for the current record.

Derived from TMySQLQuery

Create
Creates an instance of a query component.

Destroy

Destroys the instance of a query.

ExecSQL
Executes the SQL statement for the query.

GetDetailLinkFields
Fills lists with the master and detail fields of the link.

ParamByName
Accesses parameter information based on a specified parameter name.

Prepare
Sends a query for optimization prior to execution.

UnPrepare
Frees the resources allocated for a previously prepared query.

In TMySQLMacroQuery

Reopen
Reopens current query.

MacroByname
Returns Macros property item which property **Name** is equal to **Value**.

8.2.1. Reopen

Reopens current query.

Syntax:

```
procedure Reopen;
```

8.2.2. MacroByname

Returns Macros property item which property **Name** is equal to **Value**.

Syntax:

```
function MacroByname(const Value : String) : TParams;
```

Description:

Use **MacroByname** to get macro definition by its name.

8.3. Events

Please see TMySQLMacroQuery events short descriptions below:

Derived from TDataSet

AfterCancel
Occurs after an application completes a request to cancel modifications to the active record.

AfterClose
Occurs after an application closes a dataset.

AfterDelete
Occurs after an application deletes a record.

AfterEdit
Occurs after an application starts editing a record.

AfterInsert
Occurs after an application inserts a new record.

AfterOpen
Occurs after an application completes opening a dataset and before any data access occurs.

AfterPost
Occurs after an application writes the active record to the database or cache returns to browse state.

AfterRefresh
Occurs after an application refreshes the data in the dataset.

AfterScroll
Occurs after an application scrolls from one record to another.

BeforeCancel
Occurs before an application executes a request to cancel changes to the active record.

BeforeClose
Occurs before an application executes a request to close the dataset.

BeforeDelete
Occurs before an application attempts to delete the active record.

BeforeEdit
Occurs before an application enters edit mode for the active record.

BeforeInsert
Occurs before an application enters insert mode.

BeforeOpen
Occurs before an application executes a request to open a dataset.

BeforePost

Occurs before an application posts changes for the active record to the database or cache.

BeforeRefresh
Occurs immediately before an application refreshes the data in the dataset.

BeforeScroll
Occurs before an application scrolls from one record to another.

OnCalcFields
Occurs when an application recalculates calculated fields.

OnDeleteError
Occurs when an application attempts to delete a record and an exception is raised.

OnEditError
Occurs when an application attempts to modify or insert a record and an exception is raised.

OnFilterRecord
Occurs each time a different record in the dataset becomes the active record and filtering is enabled.

OnNewRecord
Occurs when an application inserts or appends a new dataset record.

OnPostError
Occurs when an application attempts to modify or insert a record and an exception is raised.

Derived from TMySQLDataSet

OnUpdateError
Occurs if an exception is generated when cached updates are applied to a database.

OnDeleting
Occurs after all checks before data deleting are passed but before an application deletes this record from the database. This event allows to cancel record deleting.

OnInserting
Occurs after all checks before data insert are passed but before an application posts changes for this new record to the database. This event allows to cancel record insertion.

OnPosting
Occurs after all checks before post are passed but before an application posts changes for the active record to the database. This event allows to cancel data modification.

In TMySQLMacroQuery

TMySQLMacroQuery has no own events

9. TMySQLMonitor

 **TMySQLMonitor** monitors dynamic SQL passed to the MySQL server.

See also: Properties, Events

9.1. Properties

Please see TMySQLMonitor properties short descriptions below:

Active
Turns the SQL monitor on and off.

TraceFlags
Indicates which database operations are traced.

Handle
Specifies the window handle that the SQL monitor uses to receive asynchronous messages.

9.1.1. Active

Turns the SQL monitor on and off.

Syntax:

```
Active: Boolean;
```

Description:

Use **Active** to turn the SQL monitor on and off. When **Active** is **True**, the SQL monitor receives an **OnSQL** event for every database operation specified by the **TraceFlags** property. When **Active** is **False**, the SQL monitor does not receive any events.

9.1.2. Handle

Specifies the window handle that the SQL monitor uses to receive asynchronous messages.

Syntax:

```
Handle: HWND;
```

Description:

The database operations that trigger **OnSQL** events are monitored by a separate thread of execution. When that thread detects a relevant database operation, it sends a Windows message to the window identified by this **Handle**.

9.1.3. TraceFlags

Indicates which database operations are traced.

Syntax:

```
type
  TMySQLTraceFlag  = (tfQPrepare,
                      tfQExecute,
                      tfQFetch,
                      tfConnect,
                      tfTransact,
                      tfMisc);
  TMySQLTraceFlags = set of TMySQLTraceFlag;
property TraceFlags: TMySQLTraceFlags;
```

Description:

Use **TraceFlags** to specify which database operations the SQL Monitor should track in an application at runtime. **TraceFlags** enables performance tuning and SQL debugging when working with remote SQL database servers.

> Normally trace options are set from the SQL Monitor rather than setting **TraceFlags** in application code.

9.2. Events

Please see TMySQLMonitor events short descriptions below:

OnSQL
Reports dynamic SQL activity on MySQL applications.

9.2.1. OnSQL

Reports dynamic SQL activity on MySQL applications.

Syntax:

```
type TSQLEvent = procedure(const Application, Database, Msg, SQL, ErrorMsg:
string;
    DataType: TMySQLTraceFlag; const ExecutedOK: boolean; EventTime: TDateTime)
```

```
of object;
OnSQL: TSQLEvent;
```

Description:

Write an **OnSQL** event handler to report dynamic SQL activity on MySQL applications.

Application
Owner application name;

Database
If SQL is executed within a specific database, this parameter indicates its name;

Msg
This parameter indicates the type of the executed operation as string - CONNECT, DISCONNECT, EXECUTE, FETCH, etc.;

SQL
This parameter contains full executed SQL statement, if such is available;

ErrorMsg
If statement executed with error, this parameter returns the error message, otherwise it is empty;

DataType
This parameter indicates the type of the executed operation as enumerated data;

ExecutedOK
This parameter indicates if operation execution succeeded;

EventTime
Indicates the time when the command passed to or from the MySQL server.

10. TMySQLQuery

 **TMySQLQuery** encapsulates a dataset with a result set that is based on an SQL statement.

Description:

Use **TMySQLQuery** to access one or more MySQL tables in a database using SQL statements. **TMySQLQuery** component is useful because it can:

- Access more than one table at a time (called a "join" in SQL).
- Automatically access a subset of rows and columns in its underlying table(s), rather than always returning all rows and columns.

TMySQLQuery provides BDE-like functionality and is fully compatible with **TDatasource** and visual DB-controls. If you need to fetch some data without displaying them you can use high-performance TMySQLDirectQuery component.

See also: Properties, Methods, Events

10.1. Properties

Please see TMySQLQuery properties short descriptions below:

Derived from TDataSet

Active
Specifies whether or not a dataset is open.

AutoCalcFields
Determines when the OnCalcFields event is triggered.

Bof
Indicates whether or not a cursor is positioned at the first record in a dataset.

Bookmark
Specifies the current bookmark in the dataset.

CachedUpdates
Does not affect on dataset behavior.

DefaultFields
Indicates whether a dataset's underlying field components are generated dynamically when the dataset is opened.

Eof
Indicates whether or not a cursor is positioned at the last record in a dataset.

FieldCount
Indicates the number of field components associated with the dataset.

FieldDefList
Points to the list of field definitions for the dataset.

FieldList
Lists the field components of a dataset.

Fields
Lists all non-aggregate field components of the dataset.

FieldValues
Provides access to the values for all fields in the active record for the dataset.

Found
Indicates whether or not moving to a different record is successful.

Modified
Indicates whether the active record is modified.

Name
Designates the name of the dataset as referenced by other components.

ObjectView
Specifies whether fields are to be stored hierarchically or flattened out in the Fields property.

SparseArrays
Determines whether a unique TField object is created for each element of an array field.

State
Indicates the current operating mode of the dataset.

Derived from TMySQLDataSet

AllowSequenced
Determines that database records can be located by sequence numbers.

AutoRefresh
Specifies whether server-generated field values are refetched automatically.

AvailableResultsetCount
Indicates count of resultsets available to fetch from multiresultset query or stored procedure. This property is useful when query or stored procedure returns more than one dataset.

BlockReadSize
Determines how many record buffers are read in each block.

CacheBlobs
Determines whether BLOB fields are cached in memory.

Database
Specifies the database component for which this dataset represents one or more tables.

Filter
Specifies the text of the current filter for a dataset.

Filtered
Specifies whether filtering is active for a dataset.

FilterOptions
Specifies whether filtering is case insensitive, and whether or not partial comparisons are permitted when filtering records.

KeySize
Specifies the size of the key for the current index of the dataset.

LastInsertID
Get last inserted value of AUTO_INCREMENT column from MySQL server.

MultiResultsetNo
Specifies the resultset to associate with component when it become active. This property is useful when query or stored procedure returns more than one dataset.

RecNo
Indicates the current record in the dataset.

RecordCount
Indicates the total number of records associated with the dataset.

RecordSize
Indicates the size of a record in the dataset.

SortFieldNames
Specifies field names and sorting order to sort opened dataset by these fields on the client side without refetching data from server.

UpdateMode
Determines how MySQL finds records when updating to an SQL database.

UpdateObject
Specifies the update object component used to update a read-only result set.

In TMySQLQuery

DataSource
Specifies the data source component from which to extract current field values to use with same-name parameters in the query SQL statement.

Handle
Specifies the cursor handle for the query.

ParamCheck
Specifies whether the parameter list for a query is regenerated if the SQL property changes at runtime.

ParamCount
Indicates the current number of parameters for the query.

Params
Contains the parameters for a query SQL statement.

Prepared
Determines whether or not a query is prepared for execution.

ProcessComments
Allows to choose what kind of comments to cut from SQL query text before sending it to server.

RequestLive

Specifies whether an application expects to receive a live result set when the query executes.

RowsAffected
Returns the number of rows operated upon by the latest query execution.

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

SQLBinary
Points to the binary data stream that represents an SQL query statement or result set.

Text
Points to the actual text of the SQL query passed to MySQL.

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

10.1.1. DataSource

Specifies the data source component from which to extract current field values to use with same-name parameters in the query's SQL statement.

Syntax:

```
property DataSource: TDataSource;
```

Description:

Set **DataSource** to automatically fill parameters in a query with fields values from another dataset. Parameters that have the same name as fields in the other dataset are filled with the field values. Parameters with names that are not the same as fields in the other dataset do not automatically get values, and must be programmatically set.

For example, if the SQL property of the **TMySQLQuery** contains the SQL statement below and the dataset referenced through **DataSource** has a *CustNo* field, the value from the current record in that other dataset is used in the *CustNo* parameter.

```
SELECT * FROM Orders O WHERE (O.CustNo = :CustNo)
```

DataSource must point to a TDataSource component linked to another dataset component; it cannot point to this query's data source component.

The dataset specified in **DataSource** must be created, populated, and opened before attempting to bind parameters. Parameters are bound by calling the query's Prepare method prior to executing the query.

DataSource is especially of use when creating a master-detail relationship between tables using a linked query. It is also of use to guarantee binding for parameters that are not already set in the **Params** property or through a call to the **ParamByName** method.

If the SQL statement used by a query does not contain parameters, or all parameters are bound by the application using the **Params** property or the **ParamByName** method, **DataSource** need not be assigned. The example below shows setting the **DataSource** property of Query2 to the data source for Query1, preparing Query2, and activating Query2.

```
with MySQLQuery2 do
begin
  DataSource := DataSource1;
  Prepare;
  Open;
end;
```

If the SQL statement in the **TMySQLQuery** is a SELECT query, the query is executed using the new field values each time the record pointer in the other dataset is changed. It is not necessary to call the Open method of the **TMySQLQuery** each time. This makes using the **DataSource** property to dynamically filter a query result set useful for establishing Master-Detail relationships. Set the **DataSource** property in the Detail query to the **TDataSource** component for the Master dataset.

If the SQL statement uses other than a SELECT query (such as INSERT or UPDATE), the parameters with the same name as fields in the other dataset still get values, but the query must be explicitly executed each time the other dataset's record pointer moves. For example, the SQL statement below uses the INSERT statement and has the parameters *Custno* and *CompanyName*.

```
INSERT INTO Customer (CustNo, Company)
  VALUES (:CustNo, :CompanyName)
```

Another dataset, **Query1** and **DataSource1**, has a *CustNo* field but no *CompanyName* field. If this dataset is used through the **DataSource** property, the *CompanyName* parameter must be programmatically assigned a value. Because **Query1** has a *CustNo* field and **Query1** is referenced through the **DataSource** property, the *CustNo* parameter automatically receives a value.

```
with MySQLQuery2 do
begin
  DataSource := DataSource1;
  ParamByName('CompanyName').AsString := Edit1.Text;
  Prepare;
  ExecSQL;
end;
```

If the SQL statement contains parameters with the same name as fields in the other dataset, do not manually set values for these parameters. Any values programmatically set, such as by using the **Params** property or the **ParamByName** method, will be overridden with automatic values. Parameters of other names must be programmatically given values. These parameters are unaffected by setting **DataSource**.

DataSource can be set at runtime or at design-time using the **Object Inspector**. At design-time, select the desired **TDataSource** from the drop-down list or type in the name.

10.1.2. Handle

Specifies the cursor handle for the query.

Syntax:

```
type HDBICur: Longint;
property Handle: HDBICur;
```

Description:

Use **Handle** only to bypass **TMySQLQuery** methods and call directly into the MySQL. Many MySQL function calls require a cursor handle parameter. **Handle** is assigned an initial value when a query is executed. If used with a call that changes the current record position, call **Resync** immediately after returning from the MySQL call.

10.1.3. ParamCheck

Specifies whether the parameter list for a query is regenerated if the SQL property changes at runtime.

```
property ParamCheck: Boolean;
```

Description:

Set **ParamCheck** to specify whether or not the **Params** property is cleared and regenerated if an application modifies the query's SQL property at runtime. By default **ParamCheck** is **True**, meaning that the **Params** property is automatically regenerated at runtime. When **ParamCheck** is **True**, the proper number of parameters is guaranteed to be generated for the current SQL statement.

This property is useful for data definition language (DDL) statements that contain parameters as part of the DDL statement and that are not parameters for the **TMySQLQuery. Set ParamCheck** to **False** to prevent these parameters from being mistaken for parameters of the **TMySQLQuery** executing the DDL statement.

An application that does not use parameterized queries may choose to set **ParamCheck** to **False**, but otherwise **ParamCheck** should be **True**.

10.1.4. ParamCount

Indicates the current number of parameters for the query.

Syntax:

```
property ParamCount: Word;
```

Description:

Inspect **ParamCount** to determine how many parameters are in the **Params** property. If the **ParamCheck** property is **True**, **ParamCount** always corresponds to the number of actual parameters in the SQL statement for the query.

See also: Example: ParamCount, DataType, StrToIntDef, AsXXX

10.1.5. Params

Contains the parameters for a query's SQL statement.

Syntax:

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

Description:

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

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

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

Example:

The following code runs an insert query to add a record for Lichtenstein into the country table.

```
MySQLQuery2.SQL.Clear;
MySQLQuery2.SQL.Add('INSERT INTO COUNTRY (NAME, CAPITAL, POPULATION)');
MySQLQuery2.SQL.Add('VALUES (:Name, :Capital, :Population)');
MySQLQuery2.Params[0].AsString := 'Lichtenstein';
MySQLQuery2.Params[1].AsString := 'Vaduz';
```

```
MySQLQuery2.Params[2].AsInteger := 420000;
MySQLQuery2.ExecSQL;
```

10.1.6. Prepared

Determines whether or not a query is prepared for execution.

Syntax:

```
property Prepared: Boolean;
```

Description:

Examine **Prepared** to determine if a query is already prepared for execution. If **Prepared** is **True**, the query is prepared, and if **Prepared** is **False**, the query is not prepared. While a query need not be prepared before execution, execution performance is enhanced if the query is prepared beforehand, particularly if it is a parameterized query that is executed more than once using the same parameter values.

> An application can change the current setting of **Prepared** to prepare or unprepare a query. If **Prepared** is **True**, setting it to **False** calls the **Unprepare** method to unprepare the query. If **Prepared** is **False**, setting it to **True** calls the **Prepare** method to prepare the query. Generally, however, it is better programming practice to call **Prepare** and **Unprepare** directly. These methods automatically update the **Prepared** property.

10.1.7. ProcessComments

Allows to choose what kind of comments to cut from SQL query text before sending it to server.

Syntax:

```
property ProcessComments : TMySQLQueryProcessComments;
```

Description:

This property is a set values that determine what kinds of comments will be deleted from SQL query text before sending it to MySQL server. The following table describes possible comments styles:

TMySQLQueryProcessComments value	Processed by default	Comments style
qpcMinusMinus	yes	All symbols from '--' until end of string are truncated.
qpcSharp	no	All symbols from '#' until end of string are truncated.
qpcSlashAsterisk	yes	C-style comments are truncated. For example '/* this is comment */'

You can set **ProcessComments** to [] (empty set) to allow MySQL server to process all kind of comments by itself. But this can cause problems if you are using query params (Params, ParamCheck, ParamCount properties).

This property can be very useful if your literal constants contains '--', '#', '/*' or '*/' symbols. Just set to remove appropriate value of **TMySQLQueryProcessComments** from **ProcessComments** property.

See also: Params property, ParamCheck property, ParamCount property

10.1.8. RequestLive

Specifies whether an application expects to receive a live result set when the query executes.

Syntax:

```
property RequestLive: Boolean;
```

Description:

Set **RequestLive** to specify whether or not the query should attempt to return a live result set to the application. **RequestLive** is **False** by default, meaning that a query always returns a read-only result set.

Set **RequestLive** to **True** to request a live result set. Setting **RequestLive** to **True** does not guarantee that a live result set is returned. MySQL returns a live result set only if the SELECT syntax of the query conforms to the syntax requirements for a live result set.

If **RequestLive** is **True**, but the syntax does not conform to the requirements, MySQL returns an error code for remote servers.

TMySQLQuery can't be "live" if meets one (or more) of following conditions:

- SQL query contains more than one tables in "from" section
- SQL query has JOINs
- SQL query doesn't use tables at all:

```
SELECT VERSION
```

- SQL query has some calculations or function calls in fields definitions:

```
SELECT field_name + 1 FROM table_name
```

- SQL query doesn't contain at least one unique indexed field.

After activation of the **TMySQLQuery** you should inspect the CanModify property to determine if the request for a live resultset succeeded.

> All multi-table queries return read-only result sets.

> Your query will not be updatable if it is multi-resultset query. This means that you will not able to edit dataset if AvailableResultsetCount property is not equal to 1.

See also: TMySQLDataset.CanModify and AvailableResultsetCount properties

10.1.9. RowsAffected

Returns the number of rows operated upon by the latest query execution.

Syntax:

```
property RowsAffected: Integer;
```

Description:

Inspect **RowsAffected** to determine how many rows were updated or deleted by the last query operation. If no rows were updated or deleted, **RowsAffected** has a value of zero. **RowsAffected** will have a value of **-1** if the execution of the SQL statement could not be executed due to an error condition. This latter situation would typically follow the raising of an exception.

10.1.10. SQL

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

Syntax:

```
property SQL: TStrings;
```

Description:

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

The **SQL** property may contain only one complete SQL statement at a time. In general, multiple "batch" statements are not allowed unless a particular server supports them.

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

> Please read this FAQ section if you want to use Unicode strings in your application: How to use Unicode data in my application?

See also: Example: SQL, ExecSQL

10.1.11. SQLBinary

Points to the binary data stream that represents an SQL query statement or result set.

Syntax:

```
property SQLBinary: PChar;
```

Description:

Do not access **SQLBinary**. It is an internal binary data stream used by the query component. To access or set the SQL statement that this query component executes, use the SQL property. To access or set the parameters used in a parameterized SQL statement, use the **Params** property.

10.1.12. Text

Points to the actual text of the SQL query passed to MySQL.

Syntax:

```
property Text: PChar;
```

Description:

Text is a read-only property that can be examined to determine the actual contents of SQL statement passed to MySQL. For parameterized queries, **Text** contains the SQL statement with parameters replaced by the parameter substitution symbol (?) in place of actual parameter values.

In general there should be no need to examine the **Text** property. To access or change the SQL statement for the query, use the SQL property. To examine or modify parameters, use the **Params** property.

10.1.13. UniDirectional

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

Syntax:

```
property UniDirectional: Boolean;
```

Description:

Set **UniDirectional** to control whether or not a cursor can move forward and backward through a result set. By default **UniDirectional** is **False**, enabling forward and backward navigation. This property may be used for BDE compatibility.

10.2. Methods

Please see TMySQLQuery methods short descriptions below:

Derived from TDataSet

ActiveBuffer
Returns a pointer to the buffer for the active record.

Append
Adds a new, empty record to the end of the dataset.

AppendRecord
Adds a new, populated record to the end of the dataset and posts it to the database.

CheckBrowseMode
Automatically posts or cancels data changes when an application changes which record in the dataset is the active record.

ClearFields
Clears the contents of all fields for the active record.

Close
Closes a dataset.

ControlsDisabled
Indicates whether data-aware controls do not update their display to reflect changes to the dataset.

CursorPosChanged
Marks the internal cursor position as invalid.

Delete
Deletes the active record and positions the cursor on the next record.

DisableControls
Disables data display in data-aware controls associated with the dataset.

Edit
Enables editing of data in the dataset.

EnableControls
Re-enables data display in data-aware controls associated with the dataset.

FieldByName
Finds a field based on its name.

FindField
Searches for a specified field in the dataset.

FindFirst
Implements a virtual method for positioning the cursor on the first record in a filtered dataset.

FindLast
Implements a virtual method for positioning the cursor on the last record in a filtered dataset.

FindNext
Implements a virtual method for positioning the cursor on the next record in a filtered dataset.

FindPrior
Implements a virtual method for positioning the cursor on the previous record in a filtered dataset.

First
Positions the cursor on the first record in the dataset.

FreeBookmark
Frees the resources allocated for a specified bookmark.

GetBookmark
Allocates a bookmark for the current cursor position in the dataset.

GetDetailDataSets
Fills a list with a dataset for every detail dataset that is not the value of a nested dataset field.

GetFieldList
Retrieves a specified set of field objects into a list.

GetFieldNames
Retrieves a list of names for all fields in a dataset.

GotoBookmark
Implements a virtual method to position the cursor on the record pointed to by a specified bookmark.

Insert
Inserts a new, empty record in the dataset.

InsertRecord
Inserts a new, populated record to the dataset and posts it to the database.

IsEmpty
Indicates whether the dataset contains no records.

IsLinkedTo
Indicates whether a dataset is linked to a specified data source.

Last
Positions the cursor on the last record in the dataset.

MoveBy
Positions the cursor on a record relative to the active record in the dataset.

Next
Positions the cursor on the next record in the dataset.

Open
Opens the dataset.

Prior
Positions the cursor on the previous record in the dataset.

Refresh
Refetches data from the database to update a dataset's view of data.

Resync
Refetches the active record and the records that precede and follow it.

SetFields
Sets the values for all fields in a record.

UpdateCursorPos
Positions the cursor on the active record.

UpdateRecord
Ensures that data-aware controls and detail datasets reflect record updates.

Derived from TMySQLDataSet

ApplyUpdates
Writes a dataset's pending cached updates to the database.

BookmarkValid
Tests the validity of a specified bookmark.

Cancel
Cancels modifications to the current record if those changes are not yet posted.

CancelUpdates
Clears all pending cached updates from the cache and restores the dataset its prior state.

CheckOpen
Checks the result of a call to the MySQL.

CloseDatabase
Closes a database connection associated with the database.

CommitUpdates
Clears the cached updates buffer.

CompareBookmarks
Indicates the relationship between two bookmarks.

FetchAll
Retrieves all records from the current cursor position to the end of the file and stores them locally.

FlushBuffers
Posts all changes that have been written to the record buffer.

GetBlobFieldData
Reads BLOB data into a buffer.

GetCurrentRecord
Retrieves the current record into a buffer.

GetFieldData
Retrieves the current value of a field into a buffer.

GetIndexInfo
Retrieves information about the current index into the index data fields of the dataset.

GetLastInsertID
Returns the ID generated for an AUTO_INCREMENT column by the previous query.

Locate

Searches the dataset for a specified record and makes that record the current record.

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

OpenDatabase
Opens the database that contains the dataset.

Post
Writes a modified record to the database.

Prepare
Sends a query for optimization prior to execution.

UnPrepare
Frees the resources allocated for a previously prepared query.

RevertRecord
Restores the current record in the dataset to an unmodified state when cached updates are enabled.

SortBy
Sorts opened dataset on client side without refetching data from server.

Translate
Converts a data string between the ANSI character set used by Delphi (and Windows), and the local code page (OEM character set).

UpdateStatus
Reports the update status for the current record.

In TMySQLQuery

Create
Creates an instance of a query component.

Destroy
Destroys the instance of a query.

ExecSQL
Executes the SQL statement for the query.

GetDetailLinkFields
Fills lists with the master and detail fields of the link.

ParamByName
Accesses parameter information based on a specified parameter name.

10.2.1. Create

Creates an instance of a query component.

Syntax:

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

Description:

Call **Create** to instantiate a query at runtime. Query components placed in forms or data modules at design time are created automatically.

Create calls its inherited **Create** constructor, creates an empty SQL statement list, an empty parameter list, sets the **OnChange** event handler for the SQL statement list, establishes a data link, sets the **RequestLive** property to **False**, sets the **ParamCheck** property to **True**, and sets the **RowsAffected** property to **-1**.

10.2.2. Destroy

Destroys the instance of a query.

Syntax:

```
destructor Destroy;
```

Description:

Do not call **Destroy** directly. Instead, call **Free**, which checks that the **TMySQLQuery** is not **nil** before calling **Destroy**.

Destroy disconnects from the server, frees the SQL statement list, the parameter list, and the data link and SQL binary storage area, and then calls its inherited destructor.

10.2.3. ExecSQL

Executes the SQL statement for the query.

Syntax:

```
procedure ExecSQL;
```

Description:

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


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

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

See also: Example: SQL, ExecSQL

10.2.4. GetDetailLinkFields

Fills lists with the master and detail fields of the link.

Syntax:

```
procedure GetDetailLinkFields(
          MasterFields,
          DetailFields: TList);
          override;
```

Description:

Creates two lists of **TFields** from the master-detail relationship between two tables; one containing the master fields, and the other containing the detail fields.

10.2.5. ParamByName

Accesses parameter information based on a specified parameter name.

Syntax:

```
function ParamByName(const Value: String): TParam;
```

Description:

Call **ParamByName** to set or use parameter information for a specific parameter based on its name. Value is the name of the parameter for which to retrieve information.

ParamByName is primarily used to set an parameter's value at runtime. For example, the following statement retrieves the current value of a parameter called "*Contact*" into an edit box:

```
Edit1.Text := MySQLQuery1.ParamByName('Contact').AsString;
```

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

10.3. Events

Please see TMySQLQuery events short descriptions below:

Derived from TDataSet

AfterCancel
Occurs after an application completes a request to cancel modifications to the active record.

AfterClose
Occurs after an application closes a dataset.

AfterDelete
Occurs after an application deletes a record.

AfterEdit
Occurs after an application starts editing a record.

AfterInsert
Occurs after an application inserts a new record.

AfterOpen
Occurs after an application completes opening a dataset and before any data access occurs.

AfterPost
Occurs after an application writes the active record to the database or cache returns to browse state.

AfterRefresh
Occurs after an application refreshes the data in the dataset.

AfterScroll
Occurs after an application scrolls from one record to another.

BeforeCancel
Occurs before an application executes a request to cancel changes to the active record.

BeforeClose
Occurs before an application executes a request to close the dataset.

BeforeDelete
Occurs before an application attempts to delete the active record.

BeforeEdit
Occurs before an application enters edit mode for the active record.

BeforeInsert
Occurs before an application enters insert mode.

BeforeOpen
Occurs before an application executes a request to open a dataset.

BeforePost
Occurs before an application posts changes for the active record to the database or cache.

BeforeRefresh
Occurs immediately before an application refreshes the data in the dataset.

BeforeScroll
Occurs before an application scrolls from one record to another.

OnCalcFields
Occurs when an application recalculates calculated fields.

OnDeleteError
Occurs when an application attempts to delete a record and an exception is raised.

OnEditError
Occurs when an application attempts to modify or insert a record and an exception is raised.

OnFilterRecord
Occurs each time a different record in the dataset becomes the active record and filtering is enabled.

OnNewRecord
Occurs when an application inserts or appends a new dataset record.

OnPostError
Occurs when an application attempts to modify or insert a record and an exception is raised.

Derived from TMySQLDataSet

OnUpdateError
Occurs if an exception is generated when cached updates are applied to a database.

OnDeleting
Occurs after all checks before data deleting are passed but before an application deletes this record from the database. This event allows to cancel record deleting.

OnInserting
Occurs after all checks before data insert are passed but before an application posts changes for this new record to the database. This event allows to cancel record insertion.

OnPosting

Occurs after all checks before post are passed but before an application posts changes for the active record to the database. This event allows to cancel data modification.

In TMySQLQuery

TMySQLQuery has no own events.

11. TMySQLStoredProc

Since v2.5.0

 **TMySQLStoredProc** provides full support for MySQL 5.0+ stored procedures.

Description:

TMySQLStoredProc allows to execute stored procedures in MySQL database. It supports IN, OUT and INOUT parameters.

TMySQLStoredProc is **TDataSet** descendant. So you can use it for store resultset fetched in stored procedure. **TMySQLStoredProc** also allows to execute stored procedures without storing resultset (for example, procedures with INSERT, UPDATE or DELETE statements).

You should use *'root'* user account to be able to work with all stored procedures. If you're using other user account, then DAC for MySQL will fetch parameters of stored procedures created by this user only.

See also: Properties, Methods, Events

11.1. Properties

Please see TMySQLStoredProc properties short descriptions below:

Derived from TDataSet

Active
Specifies whether or not a dataset is open.

AutoCalcFields
Determines when the **OnCalcFields** event is triggered.

Bof
Indicates whether or not a cursor is positioned at the first record in a dataset.

Bookmark
Specifies the current bookmark in the dataset.

CachedUpdates
Does not affect on dataset behavior.

DefaultFields
Indicates whether a dataset's underlying field components are generated dynamically when the dataset is opened.

Eof
Indicates whether or not a cursor is positioned at the last record in a dataset.

FieldCount
Indicates the number of field components associated with the dataset.

FieldDefList
Points to the list of field definitions for the dataset.

FieldList
Lists the field components of a dataset.

Fields
Lists all non-aggregate field components of the dataset.

FieldValues
Provides access to the values for all fields in the active record for the dataset.

Found
Indicates whether or not moving to a different record is successful.

Modified
Indicates whether the active record is modified.

Name
Designates the name of the dataset as referenced by other components.

ObjectView
Specifies whether fields are to be stored hierarchically or flattened out in the Fields property.

SparseArrays
Determines whether a unique **TField** object is created for each element of an array field.

State
Indicates the current operating mode of the dataset.

Derived from TMySQLDataSet

AllowSequenced
Determines that database records can be located by sequence numbers.

AutoRefresh

Specifies whether server-generated field values are refetched automatically.

AvailableResultsetCount
Indicates count of resultsets available to fetch from multiresultset query or stored procedure. This property is useful when query or stored procedure returns more than one dataset.

BlockReadSize
Determines how many record buffers are read in each block.

CacheBlobs
Determines whether BLOB fields are cached in memory.

Database
Specifies the database component for which this dataset represents one or more tables.

Filter
Specifies the text of the current filter for a dataset.

Filtered
Specifies whether filtering is active for a dataset.

FilterOptions
Specifies whether filtering is case insensitive, and whether or not partial comparisons are permitted when filtering records.

KeySize
Specifies the size of the key for the current index of the dataset.

LastInsertID
Get last inserted value of AUTO_INCREMENT column from MySQL server

RecNo
Indicates the current record in the dataset.

RecordCount
Indicates the total number of records associated with the dataset.

RecordSize
Indicates the size of a record in the dataset.

SortFieldNames
Specifies field names and sorting order to sort opened dataset by these fields on the client side without refetching data from server.

UpdateMode
Determines how MySQL finds records when updating to an SQL database.

UpdateObject
Specifies the update object component used to update a read-only result set.

In TMySQLStoredProc

MultiResultsetNo

Specifies the resultset to associate with component when it become active. This property is useful when query or stored procedure returns more than one dataset.

Params
Parameters of stored procedure.

ParamsCount
Number of parameters in **Params** list.

ProcedureName
Name of stored procedure.

RoutineType
Specifies which kind of routine must be used.

11.1.1. Params

Parameters of stored procedure.

Syntax:

```
property Params : TMySQLSPParams;
```

Description:

Use **Params** property to access stored procedure parameters list. This list is being filled right after ProcedureName property change. If you need to reload parameters list from server in design time just remove all current parameters from list and close editor. When you open editor next time parameters list is refilled.

See also: ParamsCount, ProcedureName

11.1.2. ParamsCount

Number of parameters in **Params** list.

Syntax:

```
property ParamsCount : integer;
```

See also: Params

11.1.3. ProcedureName

Name of stored procedure.

Syntax:

```
property ProcedureName: string;
```

Description:

ProcedureName property is used to set up name of stored procedure for execute, right after its value changes Params property is filled with stored procedure parameters list.

See also: Params

11.1.4. RoutineType

Specifies which kind of routine must be used.

Syntax:

```
properties RoutineType: TMySQLRoutineType;
```

```
TMySQLRoutineType = (rtProcedure, rtFunction);
```

Description:

RoutineType property is used to specify which kind of routine must be used. It can be a stored procedures or a functions. This determines the syntax for routine call. Depending on **RoutineType** the **ProcedureName** property is changing and shows only available routines.

11.2. Methods

Please see TMySQLStoredProc methods short descriptions below:

Derived from TDataSet

ActiveBuffer
Returns a pointer to the buffer for the active record.

Append
Adds a new, empty record to the end of the dataset.

AppendRecord
Adds a new, populated record to the end of the dataset and posts it to the database.

CheckBrowseMode
Automatically posts or cancels data changes when an application changes which record in the dataset is the active record.

ClearFields
Clears the contents of all fields for the active record.

Close
Closes a dataset.

ControlsDisabled
Indicates whether data-aware controls do not update their display to reflect changes to the dataset.

CursorPosChanged
Marks the internal cursor position as invalid.

Delete
Deletes the active record and positions the cursor on the next record.

DisableControls
Disables data display in data-aware controls associated with the dataset.

Edit
Enables editing of data in the dataset.

EnableControls
Re-enables data display in data-aware controls associated with the dataset.

FieldByName
Finds a field based on its name.

FindField
Searches for a specified field in the dataset.

FindFirst
Implements a virtual method for positioning the cursor on the first record in a filtered dataset.

FindLast
Implements a virtual method for positioning the cursor on the last record in a filtered dataset.

FindNext
Implements a virtual method for positioning the cursor on the next record in a filtered dataset.

FindPrior
Implements a virtual method for positioning the cursor on the previous record in a filtered dataset.

First
Positions the cursor on the first record in the dataset.

FreeBookmark
Frees the resources allocated for a specified bookmark.

GetBookmark
Allocates a bookmark for the current cursor position in the dataset.

GetDetailDataSets
Fills a list with a dataset for every detail dataset that is not the value of a nested dataset field.

GetFieldList
Retrieves a specified set of field objects into a list.

GetFieldNames
Retrieves a list of names for all fields in a dataset.

GotoBookmark
Implements a virtual method to position the cursor on the record pointed to by a specified bookmark.

Insert
Inserts a new, empty record in the dataset.

InsertRecord
Inserts a new, populated record to the dataset and posts it to the database.

IsEmpty
Indicates whether the dataset contains no records.

IsLinkedTo
Indicates whether a dataset is linked to a specified data source.

Last
Positions the cursor on the last record in the dataset.

MoveBy
Positions the cursor on a record relative to the active record in the dataset.

Next
Positions the cursor on the next record in the dataset.

Open
Opens the dataset.

Prior
Positions the cursor on the previous record in the dataset.

Refresh
Refetches data from the database to update a dataset's view of data.

Resync
Refetches the active record and the records that precede and follow it.

SetFields
Sets the values for all fields in a record.

UpdateCursorPos
Positions the cursor on the active record.

UpdateRecord
Ensures that data-aware controls and detail datasets reflect record updates.

Derived from TMySQLDataSet

ApplyUpdates
Writes a dataset's pending cached updates to the database.

BookmarkValid
Tests the validity of a specified bookmark.

Cancel
Cancels modifications to the current record if those changes are not yet posted.

CancelUpdates
Clears all pending cached updates from the cache and restores the dataset its prior state.

CheckOpen
Checks the result of a call to the MySQL.

CloseDatabase
Closes a database connection associated with the database.

CommitUpdates
Clears the cached updates buffer.

CompareBookmarks
Indicates the relationship between two bookmarks.

FetchAll
Retrieves all records from the current cursor position to the end of the file and stores them locally.

FlushBuffers
Posts all changes that have been written to the record buffer.

GetBlobFieldData
Reads BLOB data into a buffer.

GetCurrentRecord
Retrieves the current record into a buffer.

GetFieldData
Retrieves the current value of a field into a buffer.

GetIndexInfo
Retrieves information about the current index into the index data fields of the dataset.

GetLastInsertID
Returns the ID generated for an AUTO_INCREMENT column by the previous query.

Locate

Searches the dataset for a specified record and makes that record the current record.

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

OpenDatabase
Opens the database that contains the dataset.

Post
Writes a modified record to the database.

RevertRecord
Restores the current record in the dataset to an unmodified state when cached updates are enabled.

SortBy
Sorts opened dataset on client side without refetching data from server.

Translate
Converts a data string between the ANSI character set used by Delphi (and Windows), and the local code page (OEM character set).

UpdateStatus
Reports the update status for the current record.

In TMySQLStoredProc

ExecProc
Executes stored procedure without storing resultset.

ParamByName
Accesses parameter information based on a specified parameter name.

RefreshParams
Rebuilds parameters list.

SetNeedRefreshParams
Sets internal flag to refetch parameters info from server by force on the next time RefreshParams call.

11.2.1. ExecProc

Executes stored routine without storing resultset.

Syntax:

```
procedure ExecProc;
```

Description:

ExecProc method is used to execute stored routine without storing a resultset. This can be useful when routine does not return a resultset, or when a resultset from the routine is not required for further processing. **ExecProc** just executes *CALL procedure_name(params) or SELECT function_name(params)* statement filled with the parameters from Params property.

If you need to store resultset from routine please use Open method or set Active property to **True**.

See also: TMySQLStoredProc.Params, TMySQLDataSet.Open, TMySQLDataSet.Active

11.2.2. ParamByName

Accesses parameter information based on a specified parameter name.

Syntax:

```
function ParamByName(const Value: String): TParam;
```

Description:

Call **ParamByName** to return parameter information for a specific parameter based on its name. **Value** is the name of the parameter for which to retrieve information. Typically **ParamByName** is used to set an input parameter's value at runtime.

Example:

The following command line assigns the value "Jane Smith" as the value for the parameter named **Contact**:

```
StoredProc1.ParamByName('Contact').AsString := 'Jane Smith';
```

See also: TMySQLStoredProc.Params

11.2.3. RefreshParams

Rebuilds parameters list.

Syntax:

```
function RefreshParams;
```

Description:

RefreshParams is called mostly internally. When **RefreshParams** is called first time parameters are cached locally. To force **RefreshParams** refetch data directly from server call **SetNeedRefreshParams** first.

See also: TMySQLStoredProc.SetNeedRefreshParams

11.2.4. SetNeedRefreshParams

Sets internal flag to refetch parameters info from server by force on the next time RefreshParams call.

Syntax:

```
function RefreshParams;
```

Description:

To force **RefreshParams** refetch data directly from server call **SetNeedRefreshParams** first.

Example:

```
StoredProc1.SetNeedRefreshParams;
StoredProc1.RefreshParams;
StoredProc1.ParamByName('Contact').AsString := 'Jane Smith';
```

See also: TMySQLStoredProc.RefreshParams

11.3. Events

Please see TMySQLStoredProc events short descriptions below:

Derived from TDataSet

AfterCancel
Occurs after an application completes a request to cancel modifications to the active record.

AfterClose
Occurs after an application closes a dataset.

AfterDelete
Occurs after an application deletes a record.

AfterEdit
Occurs after an application starts editing a record.

AfterInsert
Occurs after an application inserts a new record.

AfterOpen
Occurs after an application completes opening a dataset and before any data access occurs.

AfterPost
Occurs after an application writes the active record to the database or cache returns to browse state.

AfterRefresh
Occurs after an application refreshes the data in the dataset.

AfterScroll
Occurs after an application scrolls from one record to another.

BeforeCancel
Occurs before an application executes a request to cancel changes to the active record.

BeforeClose
Occurs before an application executes a request to close the dataset.

BeforeDelete
Occurs before an application attempts to delete the active record.

BeforeEdit
Occurs before an application enters edit mode for the active record.

BeforeInsert
Occurs before an application enters insert mode.

BeforeOpen
Occurs before an application executes a request to open a dataset.

BeforePost
Occurs before an application posts changes for the active record to the database or cache.

BeforeRefresh
Occurs immediately before an application refreshes the data in the dataset.

BeforeScroll
Occurs before an application scrolls from one record to another.

OnCalcFields
Occurs when an application recalculates calculated fields.

OnDeleteError
Occurs when an application attempts to delete a record and an exception is raised.

OnEditError
Occurs when an application attempts to modify or insert a record and an exception is raised.

OnFilterRecord
Occurs each time a different record in the dataset becomes the active record and filtering is enabled.

OnNewRecord
Occurs when an application inserts or appends a new dataset record.

OnPostError
Occurs when an application attempts to modify or insert a record and an exception is raised.

Derived from TMySQLDataSet

OnUpdateError
Occurs if an exception is generated when cached updates are applied to a database.

OnDeleting
Occurs after all checks before data deleting are passed but before an application deletes this record from the database. This event allows to cancel record deleting.

OnInserting
Occurs after all checks before data insert are passed but before an application posts changes for this new record to the database. This event allows to cancel record insertion.

OnPosting
Occurs after all checks before post are passed but before an application posts changes for the active record to the database. This event allows to cancel data modification.

In TMySQLStoredProc

TMySQLStoredProc has no own events.

12. TMySQLTable

 **TMySQLTable** encapsulates a database table.

Description:

Use **TMySQLTable** to access data in a single database table. **TMySQLTable** provides direct access to every record and field in an underlying database table. **TMySQLTable** can also work with a subset of records within a database table using ranges and filters.

At design time, create, delete, update, or rename the database table connected to a **TMySQLTable** by right-clicking on the **TMySQLTable** and using the pop-up menu.

See also: Properties, Methods, Events

12.1. Properties

Please see TMySQLTable properties short descriptions below:

Derived from TDataSet

Active
Specifies whether or not a dataset is open.

AutoCalcFields
Determines when the **OnCalcFields** event is triggered.

Bof
Indicates whether or not a cursor is positioned at the first record in a dataset.

Bookmark
Specifies the current bookmark in the dataset.

CachedUpdates
Does not affect on dataset behavior.

DefaultFields
Indicates whether a dataset's underlying field components are generated dynamically when the dataset is opened.

Eof
Indicates whether or not a cursor is positioned at the last record in a dataset.

FieldCount
Indicates the number of field components associated with the dataset.

FieldDefList
Points to the list of field definitions for the dataset.

FieldList
Lists the field components of a dataset.

Fields
Lists all non-aggregate field components of the dataset.

FieldValues
Provides access to the values for all fields in the active record for the dataset.

Found
Indicates whether or not moving to a different record is successful.

Modified
Indicates whether the active record is modified.

Name
Designates the name of the dataset as referenced by other components.

ObjectView
Specifies whether fields are to be stored hierarchically or flattened out in the Fields property.

SparseArrays
Determines whether a unique **TField** object is created for each element of an array field.

State
Indicates the current operating mode of the dataset.

Derived from TMySQLDataSet

AllowSequenced
Determines that database records can be located by sequence numbers.

AutoRefresh
Specifies whether server-generated field values are refetched automatically.

AvailableResultsetCount
Indicates count of resultsets available to fetch from multiresultset query or stored procedure. This property is useful when query or stored procedure returns more than one dataset.

BlockReadSize
Determines how many record buffers are read in each block.

CacheBlobs
Determines whether BLOB fields are cached in memory.

Database
Specifies the database component for which this dataset represents one or more tables.

Filter
Specifies the text of the current filter for a dataset.

Filtered
Specifies whether filtering is active for a dataset.

FilterOptions
Specifies whether filtering is case insensitive, and whether or not partial comparisons are permitted when filtering records.

KeySize
Specifies the size of the key for the current index of the dataset.

LastInsertID
Get last inserted value of AUTO_INCREMENT column from MySQL server.

RecNo
Indicates the current record in the dataset.

RecordCount
Indicates the total number of records associated with the dataset.

RecordSize
Indicates the size of a record in the dataset.

SortFieldNames
Specifies field names and sorting order to sort opened dataset by these fields on the client side without refetching data from server.

UpdateMode
Determines how MySQL finds records when updating to an SQL database.

UpdateObject
Specifies the update object component used to update a read-only result set.

In TMySQLTable

BatchModify
Insert, update or delete records without refetching data from database after every operation.

CanModify
Indicates whether an application can insert, edit, and delete data in a table.

DataSource
Provides read-only access to the data source for the master dataset when this table is the detail of a master/detail relationship.

DefaultIndex
Specifies if the data in the table should be ordered on a default index when opened.

Exists
Indicates whether the underlying database table exists.

FieldDefs
Points to the list of field definitions for the dataset.

Handle
Specifies the cursor handle for the dataset.

IndexDefs
Contains information about the indexes for a table.

IndexFieldCount
Indicates the number of fields that comprise the current key.

IndexFieldNames
Lists the columns to use as an index for a table.

IndexFields
Lists the fields of the current index.

IndexName
Identifies a secondary index for the table.

KeyExclusive
Specifies how the upper and lower boundaries for a range should be interpreted.

KeyFieldCount
Specifies the number of fields to use when conducting a partial key search on a multi-field key.

Limit
Specifies the number of rows need to be fetched from table.

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

MasterSource
Specifies the name of the data source for a dataset to use as a master table in establishing a detail-master relationship between this table and another one.

Offset
Specifies the row number from which data to be fetched from table.

ReadOnly
Specifies whether a table is read-only for this application.

ReopenOnIndexChange
Enables or disables client-side dataset sorting after switching to another index.

StoreDefs
Indicates whether the table's field and index definitions persist with the data module or form.

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

12.1.1. BatchModify

Insert, update or delete records without refetching data from database after every operation.

Syntax:

```
property BatchModify: Boolean;
```

Description:

If you want implement batch Insert, Delete or Update operations, we suggest you to use **BatchModify** property.

This property allows you increase operation speed. When you set **BatchModify** to **True**, DAC for MySQL doesn't fetch data from database after each Insert or Update operations.


Use **TMySQLQuery** for batch Insert, Update or Delete operations.

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I : Integer;
begin
   MySQLTable1.BatchModify := True;
   MySQLTable1.DisableControls;
   for I := 1 to 1000 do
   begin
       MySQLTable1.Insert;
       MySQLTable1.FieldByName('ID').AsInteger := I;
       MySQLTable1.FieldByName('Name').AsString := 'name'+IntToStr(I);
       MySQLTable1.Post;
   end;
   MySQLTable1.BatchModify := False;
   MySQLTable1.EnableControls;
end;
```

12.1.2. CanModify

Indicates whether an application can insert, edit, and delete data in a table.

Syntax:

```
property CanModify: Boolean;
```

Description:

Check the status of **CanModify** to determine if an application can modify a dataset in any way. If **CanModify** is **True**, the dataset can be modified. If **CanModify** is **False**, the table is read-only.

CanModify is set automatically when an application opens a table. If the **ReadOnly** property of a table component is **True**, then **CanModify** is set to **False**.

CanModify can also be **False** because:

- Another application currently has exclusive write access to the table.
- The table is read-only by database design.

Even if **CanModify** is **True**, it is not a guarantee that a user will be able to insert or update records in a table. Other factors may come in to play, for example, SQL access privileges.

12.1.3. DataSource

Provides read-only access to the data source for the master dataset when this table is the detail of a master/detail relationship.

Syntax:

```
property DataSource: TDataSource;
```

Description:

DataSource is a read-only version of the **MasterSource** property.

12.1.4. DefaultIndex

Specifies if the data in the table should be ordered on a default index when opened.

Syntax:

```
property DefaultIndex: Boolean;
```

Description:

When **DefaultIndex** is **True**, MySQL attempts to order the data based on the primary key or a unique index when opening the table. **DefaultIndex** default set is **True**.

12.1.5. Exists

Indicates whether the underlying database table exists.

Syntax:

```
property Exists: Boolean;
```

Description:

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

Example:

The following example shows how to create a table.

```
// Don't overwrite an existing table
if not MySQLTable1.Exists then
 begin
  with MySQLTable1 do begin
// The Table component must not be active
    Active := False;
// First, describe the type of table and give
// it a name
    DatabaseName := 'DBDEMOS';
    TableName := 'CustInfo';
// Describe the fields in the table
    with FieldDefs do begin
      Clear;
      with AddFieldDef do begin
        Name := 'Field1';
        DataType := ftInteger;
        Required := True;
      end;
      with AddFieldDef do begin
        Name := 'Field2';
        DataType := ftString;
        Size := 30;
      end;
     end;
// Describe any indexes
    with IndexDefs do begin
      Clear;
      with AddIndexDef do begin
        Name := '';
        Fields := 'Field1';
        Options := [ixPrimary];
       end;
      with AddIndexDef do begin
        Name := 'Fld2Indx';
        Fields := 'Field2';
        Options := [ixCaseInsensitive];
      end;
    end;
// Call the CreateTable method to create the table
    CreateTable;
  end;
end;
```

12.1.6. FieldDefs

Points to the list of field definitions for the dataset.

Syntax:

```
property FieldDefs: TFieldDefs;
```

Description:

FieldDefs lists the field definitions for a dataset. While an application can examine **FieldDefs** to explore the field definitions for a dataset, it should not change these definitions unless creating a new table with **CreateTable** or **CreateDataSet**.

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

If the dataset includes object field descendants, **FieldDefs** represents a hierarchical view of the data, meaning that the definitions include object field definitions. To determine the definitions in a flattened view, use **FieldDefList** instead.

12.1.7. Handle

Specifies the cursor handle for the dataset.

Syntax:

```
type HDBICur: Longint;
property Handle: HDBICur;
```

Description:

Use **Handle** only to bypass **TMySQLTable** methods and call directly into the MySQL. Many function calls require a cursor handle parameter. **Handle** is assigned an initial value when a dataset is opened. If used with a call that changes the current record position, call **Resync** immediately after returning from the call.

12.1.8. IndexDefs

Contains information about the indexes for a table.

```
property IndexDefs: TIndexDefs;
```

Description:

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

Ordinarily, an application accesses or specifies indexes at runtime through the IndexFieldNames or **IndexFields** properties.

If **IndexDefs** is updated or manually edited, the **StoreDefs** property becomes **True**.

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

See also: Example: IndexDefs,Update,Count,Items,IndexName,Fields,Name

12.1.9. IndexFieldCount

Indicates the number of fields that comprise the current key.

Syntax:

```
property IndexFieldCount: Integer;
```

Description:

Examine **IndexFieldCount** to determine the number of fields that comprise the current index. For indexes based on a single column, **IndexFieldCount** returns **1**. For multi-column indexes, **IndexFieldCount** indicates the number of fields upon which the index is based.

See also: Example: IndexFields,IndexFieldCount

12.1.10. IndexFieldNames

Lists the columns to use as an index for a table.

Syntax:

```
property IndexFieldNames: String;
```

Description:

Use **IndexFieldNames** as an alternative method of specifying the index to use for a table. In **IndexFieldNames**, specify the name of each column to use as an index for a table. Ordering of column names is significant. Separate names with semicolon.

The **IndexFieldNames** and IndexName properties are mutually exclusive. Setting one clears the other.

You can control whether to sort data on client-side or on server-side by using ReopenOnIndexChange property.

See also: IndexName, ReopenOnIndexChange properties

12.1.11. IndexFields

Lists the fields of the current index.

Syntax:

```
property IndexFields: [Index: Integer]: TField;
```

Description:

IndexFields is a zero-based array of field objects, each of which corresponds to a field in the current index. Index is an ordinal value indicating the position of a field in the index. The first field in the index is *IndexFields[0]*, the second is *IndexFields[1]*, and so on.

Do not set **IndexFields** directly. Instead use the **IndexFieldNames** property to order datasets on the fly at runtime.

See also: Example: IndexFields,IndexFieldCount

12.1.12. IndexName

Identifies a secondary index for the table.

Syntax:

```
property IndexName: String;
```

Description:

Use **IndexName** to specify an alternative index for a table. If **IndexName** is empty, a table's sort order is based on its default index.

If **IndexName** contains a valid index name, then that index determines the sort order of records.

> IndexFieldNames and **IndexName** are mutually exclusive. Setting one clears the other.

This example uses the **IndexName** property to sort the records in a table on the *CustNo* and *OrderNo* fields.

```
MySQLTable1.Active := False;
// Get the current available indexes
MySQLTable1.IndexDefs.Update;
// Find one which combines customer number ('CustNo') and
// order number ('OrderNo')
for I := 0 to MySQLTable1.IndexDefs.Count - 1 do
  if MySQLTable1.IndexDefs.Items[I].Fields = 'CustNo;OrderNo' then
// Set that index as the current index for the table
    MySQLTable1.IndexName := MySQLTable1.IndexDefs.Items[I].Name;
MySQLTable1.Active := True;
```

> You can control whether to sort data on client-side or on server-side by using ReopenOnIndexChange property.

See also: IndexFieldNames, ReopenOnIndexChange properties

12.1.13. KeyExclusive

Specifies how the upper and lower boundaries for a range should be interpreted.

Syntax:

```
property KeyExclusive: Boolean;
```

Description:

Use **KeyExclusive** to specify whether a range includes or excludes the records that match the starting and ending values of the range. By default, **KeyExclusive** is **False** meaning that matching values are included.

To restrict a range to those records that are greater than the specified starting value and less than the specified ending value, set **KeyExclusive** to **True**.

Example:

```
// Limit the range from 1351 to 1356,
// including 1351 but excluding 1356
with MySQLTable1 do
begin
// Set the beginning key
  EditRangeStart;
  IndexFields[0].AsString := '1351';
// Include 1351 in the range.
// Note that KeyExclusive applys to the range start
// because of the call to EditRangeStart
  KeyExclusive := False;
// Set the ending key
  EditRangeEnd;
  IndexFields[0].AsString := '1356';
// Exclude 1356 from the range
// Note that KeyExclusive now applys to the range end
// because of the call to EditRangeEnd
  KeyExclusive := True;
// Tell the table to establish the range
  ApplyRange;
end;
```

12.1.14. KeyFieldCount

Specifies the number of fields to use when conducting a partial key search on a multi-field key.

Syntax:

```
property KeyFieldCount: Integer;
Description:
```

Description:

Use **KeyFieldCount** to limit a search based on a multi-field key to a consecutive sub-set of those fields. For example, if the primary key for a dataset consists of three-fields, a partial-key search can be conducted using only the first field in the key by setting **KeyFieldCount** to **1**. If **KeyFieldCount** is **0**, the dataset searches on all fields in the key.

> Searches are only conducted based on consecutive key fields beginning with the first field in the key. For example if a key consists of three fields, an application can set **KeyFieldCount** to **1** to search on the first field, **2** to search on the first and second fields, or **3** to search on all fields. By default **KeyFieldCount** is initially set to include all fields in a search.

12.1.15. Limit

Specifies the number of rows need to be fetched from table.

Syntax:

```
property Limit: Integer .... default -1;
```

Description:

If **Limit** > **-1** SQL Query is prepared for fetching not more than **Limit** rows (including **0**).

12.1.16. MasterFields

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

Syntax:

```
property MasterFields: String;
```

Description:

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

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

Each time the current record in the master table changes, the new values in those fields are used to select corresponding records in this table for display.

> At design time, use the **Field Link designer** to establish the master-detail relationship between two tables.

See also: Example: MasterSource, MasterFields

12.1.17. MasterSource

Specifies the name of the data source for a dataset to use as a master table in establishing a detail-master relationship between this table and another one.

Syntax:

```
property MasterSource: TDataSource;
```

Description:

Use **MasterSource** to specify the name of the data source component whose **DataSet** property identifies a dataset to use as a master table in establishing a detail-master relationship between this table and another one.

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

After setting the **MasterSource** property, specify which fields to use in the master table by setting the **MasterFields** property. At runtime each time the current record in the master table changes, the new values in those fields are used to select corresponding records in this table for display.

> At design time, use the **Field Link designer** to establish the master-detail relationship between two tables.

Example:

Suppose you have a master table named *Customer* that contains a *CustNo* field, and you also have a detail table named *Orders* that also has a *CustNo* field. To display only those records in Orders that have the same *CustNo* value as the current record in *Customer*, write this code:

```
Orders.MasterSource := 'CustSource';
Orders.MasterFields := 'CustNo';
```

If you want to display only the records in the detail table that match more than one field value in the master table, specify each field and separate them with a semicolon.

```
Orders.MasterFields := 'CustNo;SaleDate';
```

See also: Example: MasterSource, MasterFields

12.1.18. Offset

Specifies the row number from which data to be fetched from table.

Syntax:

```
property Offset: Integer .... default 0;
```

Description:

Query is prepared for fetching not more than **Limit** rows (including **0**). Rows from table are fetched starting from **Offset** row.

Example:

```
Limit  := 100;
Offset := 100;
```

Query is prepared for fetching 100 rows starting with row number 100.

> Setting **Offset** is meaningless without setting **Limit** property either. Default **Limit** value **-1** will produce syntax error in other case.

12.1.19. ReadOnly

Specifies whether a table is read-only for this application.

Syntax:

```
property ReadOnly: Boolean;
```

Description:

Use the **ReadOnly** property to prevent users from updating, inserting, or deleting data in the table. By default, **ReadOnly** is **False**, meaning users can potentially alter a table's data.

> Even if ReadOnly is **False**, users may not be able to modify or add data to a table. Other factors, such as insufficient SQL privileges for the application or its current user may prevent successful alterations.

To guarantee that users cannot modify or add data to a table:

- Set the **Active** property to **False**.
- Set **ReadOnly** to **True**.

When **ReadOnly** is **True**, the table's **CanModify** property is **False**.

12.1.20. ReopenOnIndexChange

Since v2.7.2

Enables or disables client-side dataset sorting after switching to another index.

Syntax:

```
property ReopenOnIndexChange: Boolean default True;
```

Description:

You can switch dataset to another index by using IndexName or IndexFieldNames properties. **ReopenOnIndexChange** property value controls whether dataset will be re-fetched from server or just resorted locally.

ReopenOnIndexChange property value	**TMySQLTable behavior**
True (default behavior)	Dataset is closed and whole resultset is re-queried from server again sorted according to selected index. This method is slower but you'll have fresh resultset with all changes on server side included. Sorting performed on server side respecting current character set and collation.
False	Dataset is not closed and current data is just re-ordered according to selected index using client-side sorting. This method is usually faster then first since there is no need to query data from server each time index is switched. This is the same action as calling SortBy method providing field names index is built on. But your data will be not updated with latest server-side changes. This is the same data fetched when opening dataset, it is just resorted in other order.

See also: IndexName, IndexFieldNames properties, SortBy method

12.1.21. StoreDefs

Indicates whether the table's field and index definitions persist with the data module or form.

Syntax:

```
property StoreDefs: Boolean;
```

Description:

If **StoreDefs** is **True**, the table's index and field definitions are stored with the data module or form. Setting **StoreDefs** to **True** makes the **CreateTable** method into a one-step procedure that creates fields, indexes, and validity checks at runtime.

StoreDefs is **False** by default. It becomes **True** whenever **FieldDefs** or **IndexDefs** is updated or edited manually; to prevent edited (or imported) definitions from being stored, reset **StoreDefs** to **False**.

12.1.22. TableName

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

Syntax:

```
property TableName: TFileName;
```

Description:

Use **TableName** to specify the name of the database table this component encapsulates. To set **TableName** to a meaningful value, the Database property should already be set. If **Database** is set at design time, then select a valid table name from the **TableName** drop-down list in the **Object Inspector**.

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

12.2. Methods

Please see TMySQLTable methods short descriptions below:

Derived from TDataSet

ActiveBuffer
Returns a pointer to the buffer for the active record.

Append
Adds a new, empty record to the end of the dataset.

AppendRecord
Adds a new, populated record to the end of the dataset and posts it to the database.

CheckBrowseMode

Automatically posts or cancels data changes when an application changes which record in the dataset is the active record.

ClearFields
Clears the contents of all fields for the active record.

Close
Closes a dataset.

ControlsDisabled
Indicates whether data-aware controls do not update their display to reflect changes to the dataset.

CursorPosChanged
Marks the internal cursor position as invalid.

Delete
Deletes the active record and positions the cursor on the next record.

DisableControls
Disables data display in data-aware controls associated with the dataset.

Edit
Enables editing of data in the dataset.

EnableControls
Re-enables data display in data-aware controls associated with the dataset.

FieldByName
Finds a field based on its name.

FindField
Searches for a specified field in the dataset.

FindFirst
Implements a virtual method for positioning the cursor on the first record in a filtered dataset.

FindLast
Implements a virtual method for positioning the cursor on the last record in a filtered dataset.

FindNext
Implements a virtual method for positioning the cursor on the next record in a filtered dataset.

FindPrior
Implements a virtual method for positioning the cursor on the previous record in a filtered dataset.

First
Positions the cursor on the first record in the dataset.

FreeBookmark
Frees the resources allocated for a specified bookmark.

GetBookmark

Allocates a bookmark for the current cursor position in the dataset.

GetDetailDataSets
Fills a list with a dataset for every detail dataset that is not the value of a nested dataset field.

GetFieldList
Retrieves a specified set of field objects into a list.

GetFieldNames
Retrieves a list of names for all fields in a dataset.

GotoBookmark
Implements a virtual method to position the cursor on the record pointed to by a specified bookmark.

Insert
Inserts a new, empty record in the dataset.

InsertRecord
Inserts a new, populated record to the dataset and posts it to the database.

IsEmpty
Indicates whether the dataset contains no records.

IsLinkedTo
Indicates whether a dataset is linked to a specified data source.

Last
Positions the cursor on the last record in the dataset.

MoveBy
Positions the cursor on a record relative to the active record in the dataset.

Next
Positions the cursor on the next record in the dataset.

Open
Opens the dataset.

Prior
Positions the cursor on the previous record in the dataset.

Refresh
Refetches data from the database to update a dataset's view of data.

Resync
Refetches the active record and the records that precede and follow it.

SetFields
Sets the values for all fields in a record.

UpdateCursorPos
Positions the cursor on the active record.

UpdateRecord

Ensures that data-aware controls and detail datasets reflect record updates.

Derived from TMySQLDataSet

ApplyUpdates
Writes a dataset's pending cached updates to the database.

BookmarkValid
Tests the validity of a specified bookmark.

Cancel
Cancels modifications to the current record if those changes are not yet posted.

CancelUpdates
Clears all pending cached updates from the cache and restores the dataset its prior state.

CheckOpen
Checks the result of a call to the MySQL.

CloseDatabase
Closes a database connection associated with the database.

CommitUpdates
Clears the cached updates buffer.

CompareBookmarks
Indicates the relationship between two bookmarks.

FetchAll
Retrieves all records from the current cursor position to the end of the file and stores them locally.

FlushBuffers
Posts all changes that have been written to the record buffer.

GetBlobFieldData
Reads BLOB data into a buffer.

GetCurrentRecord
Retrieves the current record into a buffer.

GetFieldData
Retrieves the current value of a field into a buffer.

GetIndexInfo
Retrieves information about the current index into the index data fields of the dataset.

GetLastInsertID
Returns the ID generated for an AUTO_INCREMENT column by the previous query.

Locate
Searches the dataset for a specified record and makes that record the current record.

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

OpenDatabase
Opens the database that contains the dataset.

Post
Writes a modified record to the database.

RevertRecord
Restores the current record in the dataset to an unmodified state when cached updates are enabled.

SortBy
Sorts opened dataset on client side without refetching data from server.

Translate
Converts a data string between the ANSI character set used by Delphi (and Windows), and the local code page (OEM character set).

UpdateStatus
Reports the update status for the current record.

InTMySQLTable

AddIndex
Creates a new index for the table.

ApplyRange
Applies a range to the dataset.

CancelRange
Removes any ranges currently in effect for the table.

Create
Creates an instance of a table component.

CreateBlobStream
Returns a **TMySQLBlobStream** object for reading or writing the data in a specified blob field.

CreateTable
Builds a new table using new structure information.

DeleteIndex
Deletes a secondary index for the table.

Destroy
Destroys the instance of a component.

EditKey
Enables modification of the search key buffer.

EditRangeEnd
Enables changing the ending value for an existing range.

EditRangeStart
Enables changing the starting value for an existing range.

EmptyTable
Deletes all records from the table.

FindKey
Searches for a record containing specified field values.

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

GetDetailLinkFields
Lists the field components that link this dataset as a detail of a master dataset.

GetIndexNames
Retrieves a list of available indexes for a table.

GetTableEngine
Return a table Engine type as string.

GotoCurrent
Synchronizes the current record for this table with the current record of a specified table component.

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

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

IsSequenced
Indicates whether the underlying database table uses record numbers to indicate the order of records.

LockTable
Locks a table.

RenameTable
Renames the table associated with this table component.

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

SetRange
Sets the starting and ending values of a range, and applies it.

SetRangeEnd
Indicates that subsequent assignments to field values specify the end of the range of rows to include in the dataset.

SetRangeStart
Indicates that subsequent assignments to field values specify the start of the range of rows to include in the dataset.

UnlockTable
Removes a previously applied lock on a table.

12.2.1. AddIndex

Creates a new index for the table.

Syntax:

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

Description:

Call **AddIndex** to create a new index for the already-existing table associated with a dataset component. The index created with this procedure is added to the database table underlying the dataset component.

Name is the name of the new index. Name must contain an index name that is valid for MySQL.

Fields is a String value containing the field or fields on which the new index will be based. If more than one field is used, separate the field names in the list with semi-colons.

Options is a set of attributes for the index. The **Options** parameter may contain any one, multiple, or none of the **TIndexOptions** constants: **ixPrimary**, **xUnique**, **ixDescending**, **ixCaseInsensitive**, and **ixExpression**.

DescFields is a string containing a list of field names, separated by semi-colons. The fields specified in **DescFields** are the fields in the new index for which the ordering will be descending. Fields in the index definition but not in the **DescFields** list use the default ascending ordering. It is possible that a single index can have fields using both ascending and descending ordering.

```
MySQLTable1.AddIndex('MostPaid',
                     'CustNo;SaleDate;AmountPaid',
                     [ixCaseInsensitive],
                     'SaleDate;AmountPaid');
```

Example:

In the example below, the **AddIndex** method is used to create an index named **NewIndex**. This index is based on two fields from the associated table, *CustNo* and *CustName*. The index **NewIndex** incorporates two index options through the **TIndexOptions** constants **ixUnique** and **ixCaseInsensitive**.

```
MySQLTable1.AddIndex('NewIndex',
                     'CustNo;CustName',
                     [ixUnique, ixCaseInsensitive]);
```

⚠ Attempting to create an index using options that are not applicable to the table type causes **AddIndex** to raise an exception.

12.2.2. ApplyRange

Applies a range to the dataset.

Syntax:

```
procedure ApplyRange;
```

Description:

Call **ApplyRange** to cause a range established with SetRangeStart and SetRangeEnd, or EditRangeStart and EditRangeEnd, to take effect. When a range is in effect, only those records that fall within the range are available to the application for viewing and editing.

See also: SetRangeStart, SetRangeEnd, EditRangeStart, EditRangeEnd, SetRange methods, EditRangeStart, EditRangeEnd, FieldByName, ApplyRange example

12.2.3. CancelRange

Removes any ranges currently in effect for the table.

Syntax:

```
procedure CancelRange;
```

Description:

Call **CancelRange** to remove a range currently applied to a table. Canceling a range re-enables access to all records in the dataset.

See also: Example: SetRange, CancelRange, Refresh

12.2.4. Create

Creates an instance of a table component.

Syntax:

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

Description:

Call **Create** to instantiate a table declared in an application if it was not placed on a form or data module at design time. **Create** calls its inherited constructor, creates an empty index definitions list, creates an empty data link, and creates an empty list of index files.

12.2.5. CreateBlobStream

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

Syntax:

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

Description:

Call **CreateBlobStream** to obtain a stream for reading data from or writing data to a binary large object (BLOB) field. The Field parameter must specify a **TBlobField** component from the **Fields** property array.

The **Mode** parameter specifies whether the stream will be used for reading, writing, or updating the contents of the field.

Blob streams are created in a specific mode for a specific field on a specific record. Applications should create a new blob stream every time the record in the dataset changes rather than reusing an existing blob stream.

See also: Example: Create, CreateBlobStream, Edit, CopyFrom

12.2.6. CreateTable

Builds a new table using new structure information.

Syntax:

```
procedure CreateTable;
```

Description:

Call **CreateTable** at runtime to create a table using this dataset's current definitions. If the table already exists, **CreateTable** overwrites the table's structure and data. To avoid overwriting an existing table, check the Exists property before calling **CreateTable**.

If the FieldDefs property contains values, these values are used to create field definitions. Otherwise the Fields property is used. One or both of these properties must contain values in order to create a database table.

If the IndexDefs property contain values, these values are used to create indexes on the table.

See also: Exists, FieldDefs, IndexDefs properties, CreateTable usage example

12.2.7. DeleteIndex

Deletes a secondary index for the table.

Syntax:

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

Description:

Call **DeleteIndex** to remove a secondary index for a table. Name is the name of the index to delete. **DeleteIndex** cannot remove a primary index.

> To delete an index, an application must first open the table and then lock it.

12.2.8. Destroy

Destroys the instance of a component.

Syntax:

```
destructor Destroy;
```

Description:

Do not call **Destroy** directly. Instead, call **Free**, which verifies that the table is not **nil** before calling **Destroy**. **Destroy** frees the index files list for the table, frees its data link, frees its index definitions, and then calls its inherited destructor.

12.2.9. EditKey

Enables modification of the search key buffer.

Syntax:

```
procedure EditKey;
```

Description:

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

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

See also: Example: EditKey, GotoKey

12.2.10. EditRangeEnd

Enables changing the ending value for an existing range.

Syntax:

```
procedure EditRangeEnd;
```

Description:

Call **EditRangeEnd** to change the ending value for an existing range. To specify an end range value, call **FieldByName** after calling **EditRangeEnd**. After assigning a new ending value, call **ApplyRange** to activate the modified range.

See also: Example: EditRangeStart, EditRangeEnd, FieldByName, ApplyRange

12.2.11. EditRangeStart

Enables changing the starting value for an existing range.

Syntax:

```
procedure EditRangeStart;
```

Description:

Call **EditRangeStart** to change the starting value for an existing range. To specify a start range value, call **FieldByName** after calling **EditRangeStart**. After assigning a new ending value, call **ApplyRange** to activate the modified range.

See also: Example: EditRangeStart, EditRangeEnd, FieldByName, ApplyRange

12.2.12. EmptyTable

Deletes all records from the table.

Syntax:

```
procedure EmptyTable;
```

Description:

The **EmptyTable** method deletes all records from the database table specified by the **DatabaseName** and **TableName** properties.

Deletion of records can fail if the user lacks sufficient privileges to perform the delete operation.

See also: Example: EmptyTable

12.2.13. FindKey

Searches for a record containing specified field values.

Syntax:

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

Description:

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

For MySQL tables, the key may correspond to a specified index in IndexName, or to a list of field names in the **TMySQLTable.IndexFieldNames** property.

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

> **Delphi 7** and prior has poor support for *int64* values in *variant* type. This means that you'll be unable to use Locate and similar methods with such fields. Lookup fields and master-detail tables will not work properly too because of their dependence on **Locate** method. Please update your IDE to BDS 2006 (or later) or don't use BIGINT and UNSIGNED INT datatypes for keys and indexes if you need to use them in lookup fields. Also you can't use variant-style properties (**FieldValues**, **AsVariant** and so on) with such fields.

12.2.14. FindNearest

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

Syntax:

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

Description:

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

For MySQL tables, the key may correspond to a specified index in **TMySQLTable.IndexName,** or to a list of field names in the **TMySQLTable.IndexFieldNames** property.

FindNearest positions the cursor either on a record that exactly matches the search criteria, or on the first record whose values are greater than those specified in the search criteria. **KeyExclusive** affects the boundary conditions of ranges and will affect the record selected by **FindNearest**.

> **Delphi 7** and prior has poor support for *int64* values in *variant* type. This means that you'll be unable to use Locate and similar methods with such fields. Lookup fields and master-detail tables will not work properly too because of their dependence on **Locate** method. Please update your IDE to BDS 2006 (or later) or don't use BIGINT and UNSIGNED INT datatypes for keys and indexes if you need to use them in lookup fields. Also you can't use variant-style properties (**FieldValues**, **AsVariant** and so on) with such fields.

See also: Example: FindNearest

12.2.15. GetDetailLinkFields

Lists the field components that link this dataset as a detail of a master dataset.

Syntax:

```
procedure GetDetailLinkFields(MasterFields, DetailFields: TList);
override;
```

Description:

GetDetailLinkFields fills two lists of **TFields** that define a master-detail relationship between this table and another (master) dataset. The **MasterFields** list is filled with fields from the master table whose values must equal the values of the fields in the **DetailFields** list. The **DetailFields** list is filled with fields from the calling dataset.

12.2.16. GetIndexNames

Retrieves a list of available indexes for a table.

Syntax:

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

Description:

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

12.2.17. GetTableEngine

Return a table Engine type as string.

Syntax:

```
function GetTableEngine : String;
```

Description:

Call to **GetTableEngine** to obtain table Engine type.

12.2.18. GotoCurrent

Synchronizes the current record for this table with the current record of a specified table component.

Syntax:

```
procedure GotoCurrent(Table: TMySQLTable);
```

Description:

Call **GotoCurrent** to synchronize the cursor position for this table with the cursor position in another dataset that uses a different data source component, but which is connected to the same underlying database table. Table is the name of the table component whose cursor position to use for synchronizing.

> This procedure works only for table components that have the same Database and **TableName** properties. Otherwise, **GotoCurrent** raises an exception.

GotoCurrent is mainly for use in applications that have two table components that are linked to the same underlying database table through different data source components. It enables an application to ensure that separate views of the data appear to be linked.

12.2.19. GotoKey

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

Syntax:

```
function GotoKey: Boolean;
```

Description:

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

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

> **Delphi 7** and prior has poor support for *int64* values in *variant* type. This means that you'll be unable to use Locate and similar methods with such fields. Lookup fields and master-detail tables will not work properly too because of their dependence on **Locate** method. Please update your IDE to BDS 2006 (or later) or don't use BIGINT and UNSIGNED INT datatypes for keys and indexes if you need to use them in lookup fields. Also you can't use variant-style properties (**FieldValues**, **AsVariant** and so on) with such fields.

See also: **Example: EditKey, GotoKey**

12.2.20. GotoNearest

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

Syntax:

```
procedure GotoNearest;
```

Description:

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

> KeyExclusive determines which records are considered part of a search range.

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

> If some field from multi-field index is not assigned with a value after EditKey call it will not be used for search.

Delphi 7 and prior has poor support for *int64* values in *variant* type. This means that you'll be unable to use Locate and similar methods with such fields. Lookup fields and master-detail tables will not work properly too because of their dependence on **Locate** method. Please update your IDE to BDS 2006 (or later) or don't use BIGINT and UNSIGNED INT datatypes for keys and indexes if you need to use them in lookup fields. Also you can't use variant-style properties (**FieldValues**, **AsVariant** and so on) with such fields.

See also: SetKey method, KeyExclusive property, SetKey,GotoNearest example

12.2.21. IsSequenced

Indicates whether the underlying database table uses record numbers to indicate the order of records.

Syntax:

```
function IsSequenced: Boolean;
```

Description:

Use **IsSequenced** to determine whether the underlying database table supports sequence numbers, or whether these are computed by the dataset component. When **IsSequenced** returns **True**, applications can safely use the **RecNo** property to navigate to records in the dataset.

12.2.22. LockTable

Locks a table.

Syntax:

```
procedure LockTable(LockType: TLockType);
type TLockType = (ltReadLock, ltWriteLock);
```

Description:

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

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

See also: UnlockTable method

12.2.23. RenameTable

Renames the table associated with this table component.

Syntax:

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

Description:

Call **RenameTable** to give a new name to the table underlying this table component. **RenameTable** renames the table and any support files.

12.2.24. SetKey

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

Syntax:

```
procedure SetKey;
```

Description:

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

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

12.2.25. SetRange

Sets the starting and ending values of a range, and applies it.

Syntax:

```
procedure SetRange(
  const StartValues, EndValues: array of const);
```

Description:

Call **SetRange** to specify a range and apply it to the dataset. **StartValues** indicates the field values that designate the first record in the range. **EndValues** indicates the field values that designate the last record in the range.

SetRange combines the functionality of **SetRangeStart**, **SetRangeEnd**, and **ApplyRange** in a single procedure call. **SetRange** performs the following functions:

- Puts the dataset into **dsSetKey** state.
- Erases any previously specified starting range values and ending range values.
- Sets the start and end range values.
- Applies the range to the dataset.

If either **StartValues** or **EndValues** has fewer elements than the number of fields in the current index, then the remaining entries are set to NULL.

> With MySQL, **SetRange** works with any columns specified in the **IndexFieldNames** property.

See also: SetRangeStart, SetRangeEnd methods

12.2.26. SetRangeEnd

Indicates that subsequent assignments to field values specify the end of the range of rows to include in the dataset.

Syntax:

```
procedure SetRangeEnd;
```

Description:

Call **SetRangeEnd** to put the dataset into **dsSetKey** state, erase any previous end range values, and set them to NULL. Subsequent key buffer field assignments can be made using the **FieldByName** method to set the ending values for a range.

After assigning end-range values to **FieldValues**, call **ApplyRange** to activate the modified range.

> With MySQL, **TMySQLTable.SetRangeEnd** works with any columns specified in the **IndexFieldNames** property.

See also: SetRangeStart, SetRange methods

12.2.27. SetRangeStart

Indicates that subsequent assignments to field values specify the start of the range of rows to include in the dataset.

Syntax:

```
procedure SetRangeStart;
```

Description:

Call **SetRangeStart** to put the dataset into **dsSetKey** state, erase any previous start range values, and set them to NULL. Subsequent field assignments to the **FieldValues** property indicate the actual set of starting values for a range.

After assigning start-range values to **FieldValues**, call **ApplyRange** to activate the modified range.

With MySQL, **TMySQLTable.SetRangeStart** works with any columns specified in the **IndexFieldNames** property.

See also: SetRange, SetRangeEnd methods

12.2.28. UnlockTable

Removes a previously applied lock on a table.

Syntax:

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

Description:

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

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

See also: LockTable method

12.3. Events

Please see TMySQLTable events short descriptions below:

Derived from TDataSet

AfterCancel
Occurs after an application completes a request to cancel modifications to the active record.

AfterClose
Occurs after an application closes a dataset.

AfterDelete
Occurs after an application deletes a record.

AfterEdit
Occurs after an application starts editing a record.

AfterInsert
Occurs after an application inserts a new record.

AfterOpen
Occurs after an application completes opening a dataset and before any data access occurs.

AfterPost
Occurs after an application writes the active record to the database or cache returns to browse state.

AfterRefresh
Occurs after an application refreshes the data in the dataset.

AfterScroll
Occurs after an application scrolls from one record to another.

BeforeCancel
Occurs before an application executes a request to cancel changes to the active record.

BeforeClose
Occurs before an application executes a request to close the dataset.

BeforeDelete
Occurs before an application attempts to delete the active record.

BeforeEdit
Occurs before an application enters edit mode for the active record.

BeforeInsert
Occurs before an application enters insert mode.

BeforeOpen
Occurs before an application executes a request to open a dataset.

BeforePost
Occurs before an application posts changes for the active record to the database or cache.

BeforeRefresh
Occurs immediately before an application refreshes the data in the dataset.

BeforeScroll
Occurs before an application scrolls from one record to another.

OnCalcFields
Occurs when an application recalculates calculated fields.

OnDeleteError
Occurs when an application attempts to delete a record and an exception is raised.

OnEditError
Occurs when an application attempts to modify or insert a record and an exception is raised.

OnFilterRecord
Occurs each time a different record in the dataset becomes the active record and filtering is enabled.

OnNewRecord
Occurs when an application inserts or appends a new dataset record.

OnPostError
Occurs when an application attempts to modify or insert a record and an exception is raised.

Derived from TMySQLDataSet

OnUpdateError
Occurs if an exception is generated when cached updates are applied to a database.

OnDeleting
Occurs after all checks before data deleting are passed but before an application deletes this record from the database. This event allows to cancel record deleting.

OnInserting
Occurs after all checks before data insert are passed but before an application posts changes for this new record to the database. This event allows to cancel record insertion.

OnPosting
Occurs after all checks before post are passed but before an application posts changes for the active record to the database. This event allows to cancel data modification.

In TMySQLTable

TMySQLTable has no own events

13. TMySQLTools

 **TMySQLTools** component allows to run MySQL diagnostic and repair operations such as **Repair**, **Check**, **Analyze**, **Optimize**, **Backup** and **Restore**.

> Please note that **TMySQLTools** component doesn't perform these operations itself: it just executes maintenance queries on server side. Please find additional details at http://dev.mysql.com/doc/refman/5.0/en/table-maintenance-sql.html

BACKUP TABLE and RESTORE TABLE queries are deprecated in current MySQL versions (http://dev.mysql.com/doc/refman/5.0/en/backup-table.html).

We strongly recommend to use TMySQLDump component to backup your data, and TMySQLBatchExecute component to restore SQL-dumps instead of using **TMySQLTools** component for Backup and Restore operations.

See also: Properties, Methods, Events, TMySQLDump, TMySQLBatchExecute components

13.1. Properties

Please see TMySQLTools properties short descriptions below:

Database
Points to TMySQLDatabase component which sets a DB to be connected with.

Directory
Sets directory on server side for backup MySQL tables to.

TableList
Sets a list of DB tables to be dumped.

CheckOption
Sets **MySQLOperation** operation parameter.

MySQLOperation
Sets operation with which tables from TableList property you want to be processed.

RepairOption
Sets MySQL Repair command parameters.

13.1.1. CheckOption

Sets **MySQLOperation** operation parameter.

Syntax:

```
CheckOption :  TCheckOption;
Type
TCheckOption = (coQuick,coFast,coMedium,coExtended,coChanged);
```

Description:

Default value is **coQuick**.

Possible parameters values:

coQuick
Don't scan the rows to check for wrong links;

coFast
Only check tables which haven't been closed properly;

coChanged
Only check tables which have been changed since last check or haven't been closed properly;

coMedium
Scan rows to verify that deleted links are okay. This also calculates a key checksum for the rows and verifies this with a calculated checksum for the keys;

coExtended
Do a full key lookup for all keys for each row. This ensures that the table is 100% consistent, but will take a long time!

13.1.2. Database

Points to TMySQLDatabase component which sets a DB to be connected with.

Syntax:

```
Database: TMySQLDatabase;
```

13.1.3. Directory

Sets directory on server side for backup MySQL tables to.

Syntax:

```
property Directory : string
```

Description:

Use **Directory** to set path to directory on server side for backup tables from TableList to. Value of **Directory** used with BACKUP TABLE MySQL statement:

```
BACKUP TABLE tbl_name [, tbl_name] ... TO '/path/to/backup/directory'
```

Example:

```
mySQLTools1.Directory := 'C:/backup/mysql';
```

See also: TableList

13.1.4. MySQLOperation

Sets operation with which tables from. TableList property you want to be processed.

Syntax:

```
MySQLOperation : TMySQLOperation;
Type
TMySQLOperation = (oOptimize, oCheck,oAnalyze,oRepair,oBackup,oRestore);
```

Description:

Default value is **oCheck**.

Possible parameters values:

oRepair
Repairs a possible corrupted table;

oCheck
Checks the table(s) for errors;

oRestore
Restores the table(s) from the backup that was made with BACKUP;

oOptimize
OPTIMIZE should be used if you have deleted a large part of a table or if you have made many changes to a table with variable-length rows;

oAnalize
Analyse and store the key distribution for the table;

oBackup
Backups the table(s).

13.1.5. RepairOption

Sets MySQL Repair command parameters.

Syntax:

```
RepairOption : TRepairOption;
Type
TRepairOption = (roQuick,roExtended);
```

Description:

Sets MySQL Repair command parameters (**roQuick** by default). Possible parameters values:

roQuick
MySQL will try to do a REPAIR of only the index tree;

roExtended
MySQL will create the index row by row instead of creating one index at a time with sorting; this may be better than sorting on fixed-length keys if you have long CHAR keys that compress very well.

13.1.6. TableList

Sets a list of DB tables to be dumped.

Syntax:

```
TableList : TStrings;
```

Description:

Use **TableList** to set a list of DB tables to be processed with desired tool(s).

13.2. Methods

Please see TMySQLTools methods short descriptions below:

Execute
Executes the command which is set in MySQLOperation.

13.2.1. Execute

Executes the command which is set in MySQLOperation.

Syntax:

```
Function Execute: Boolean;
```

13.3. Events

Please see TMySQLTools events short descriptions below:

OnError
Occurs on error while MySQL operation is executed by Execute

OnSuccess
Occurs on successful table processing completion.

13.3.1. OnError

Occurs on error while MySQL operation is executed by Execute

Syntax:

```
type TErrorEvent = procedure(TableName,ErrorMessage: String) of object;
OnError: TErrorEvent;
```

Description:

Create **OnError** event handler to get information about MySQL operation errors.

TableName
Contains the name of the table which was in processing when an error occurs;

ErrorMessage
Contains error message text.

13.3.2. OnSuccess

Occurs on successful table processing completion.

Syntax:

```
type TSuccessEvent = procedure(TableName,Status: String) of object;
OnSuccess : TSuccessEvent;
```

Description:

Create **OnSuccess** event handler to get information about successful table processing completion.

TableName
The name of the table just processed;

Status
Table processing completion status.

14. TMySQLUpdateSQL


TMySQLUpdateSQL applies updates on behalf of queries or stored procedures that can't post updates directly.

Description:

Use a **TMySQLUpdateSQL** object to provide SQL statements used to update read-only datasets represented by TMySQLQuery component. A dataset is read-only either by design or circumstance. If a dataset is read-only by design, the application itself does not provide a user interface for updating data, but may institute a programmatic scheme behind the scenes. If a dataset is read-only by circumstance, it indicates that the MySQL returned a read-only result set. This usually happens for queries made against multiple tables. Such queries are, by SQL-92 definitions, read-only.

TMySQLUpdateSQL provides a mechanism for circumventing what some developers consider an SQL-92 limitation. It enables a developer to provide INSERT, UPDATE, and DELETE statements for performing separate update queries on otherwise read-only result sets in such a manner that the separate update queries are transparent to the end user.

In practical application, a **TMySQLUpdateSQL** object is placed on a data module or form, and linked to a TMySQLQuery component through that component's UpdateObject property. If the UpdateObject property points to a valid **TMySQLUpdateSQL** object, the SQL statements belonging to the update object are automatically applied when updates are applied.

BLOB parameters (including MEMO fields, GRAPHIC fields etc.) support were added in the version 2.6.2

See also: Properties, Methods

14.1. Properties

Please see TMySQLUpdateSQL properties short descriptions below:

DataSet
Identifies the dataset to which a **TMySQLUpdateSQL** component belongs.

DeleteSQL
Specifies the SQL DELETE statement to use when applying a cached deletion of a record.

InsertSQL
Specifies the SQL INSERT statement to use when applying a cached insertion of a record.

ModifySQL
Specifies the SQL UPDATE statement to use when applying an update to a record and cached updates is enabled.

Query
Returns the query object used to perform a specified kind of update.

RefreshRecordSQL
Specifies an SQL statement to use to refresh a single record.

SQL
Returns a specified SQL statement used when applying cached updates.

14.1.1. DataSet

Identifies the dataset to which a **TMySQLUpdateSQL** component belongs.

Syntax:

```
property DataSet: TMySQLDataSet;
```

Description:

At design time, setting the dataset object's **UpdateObject** property automatically sets the **DataSet** property of the specified **TMySQLUpdateSQL** object. An application should only need to set this property if it creates a new update component at run time.

14.1.2. DeleteSQL

Specifies the SQL DELETE statement to use when applying a cached deletion of a record.

Syntax:

```
property DeleteSQL: TStrings;
```

Description:

Set **DeleteSQL** to the SQL DELETE statement to use when applying a deletion to a record. Statements can be parameterized queries. To create a DELETE statement at design time, use the **UpdateSQL** editor to create statements, such as:

```
DELETE FROM "Country" WHERE Name = :OLD_Name
```

At run time, an application can write a statement directly to this property to set or change the DELETE statement.

> As the example illustrates, **DeleteSQL** supports an extension to normal parameter binding. To retrieve the value of a field as it exists prior to application of cached updates, the field name with *'OLD_'*. This is especially useful when doing field comparisons in the WHERE clause of the statement.

14.1.3. InsertSQL

Specifies the SQL INSERT statement to use when applying a cached insertion of a record.

Syntax:

```
property InsertSQL: TStrings;
```

Description:

Set **InsertSQL** to the SQL INSERT statement to use when applying an insertion to a dataset. Statements can be parameterized queries. To create a INSERT statement at design time, use the **UpdateSQL** editor to create statements, such as:

```
INSERT INTO "Country" (Name, Capital, Continent)
  VALUES (:Name, :Capital, :Continent)
  WHERE :OLD_Name = "Rangoon"
```

At run time, an application can write a statement directly to this property to set or change the INSERT statement.

> As the example illustrates, **InsertSQL** supports an extension to normal parameter binding. To retrieve the value of a field as it exists prior to application of cached updates, the field name with *'OLD_'*. This is especially useful when doing field comparisons in the WHERE clause of the statement.

t

14.1.4. ModifySQL

Specifies the SQL UPDATE statement to use when applying an update to a record and cached updates is enabled.

Syntax:

```
property ModifySQL: TStrings;
```

Description:

Set **ModifySQL** to the SQL UPDATE statement to use when applying an updated record to a dataset. Statements can be parameterized queries. To create a UPDATE statement at design time, use the **UpdateSQL** editor to create statements, such as:

```
UPDATE "Country"
SET Name = :Name, Capital = :Capital,
Continent = :Continent
WHERE Name = :OLD_Name
```

At run time, an application can write a statement directly to this property to set or change the UPDATE statement.

> As the example illustrates, **ModifySQL** supports an extension to normal parameter binding. To retrieve the value of a field as it exists prior to application of cached updates, the field name with '*OLD_*'. This is especially useful when doing field comparisons in the WHERE clause of the statement.

14.1.5. Query

Returns the query object used to perform a specified kind of update.

Syntax:

```
property Query[UpdateKind: TUpdateKind]: TMySQLQuery;
```

Description:

Query is a read-only property that provides a reference to the internal **TMySQLQuery** that executes the SQL that applies the cached data updates. Use properties and methods of **TMySQLQuery** to work with this reference.

Using one of the **TUpdateKind** constants as an index for the Query property, the internal **TMySQLQuery** object will have the SQL specified in the corresponding update SQL property: **DeleteSQL**, **InsertSQL**, or **ModifySQL**. **UpdateKind** can be one of the following:

Value	Meaning
ukDelete	Return the query object used to execute DELETE statements (DeleteSQL).
ukInsert	Return the query object used to execute INSERT statements (InsertSQL).
ukModify	Return the query object used to execute UPDATE statements (ModifySQL).

Each query object executes a particular kind of SQL statement. The contents of the SQL statements executed by these objects can be accessed directly using the **ModifySQL**, **InsertSQL**, and **DeleteSQL** properties.

The main purpose of **Query** is to provide a way for an application to set the properties for an update query object or to call the query object's methods.

If a particular kind of update statement is not provided, then its corresponding query object is **nil**. For example, if an application does not provide an SQL statement for the **DeleteSQL** property, then setting *Query[ukDelete]* returns **nil**.

14.1.6. RefreshRecordSQL

Specifies an SQL statement to use to refresh a single record.

Syntax:

```
property RefreshRecordSQL: TStrings;
```

Description:

Set **RefreshRecordSQL** to an SQL statement to use when refresh a single record from a database. Statements can be parameterized queries. To create a statement at design time, use the **RefreshRecordSQL** editor to create statements, such as:

```
SELECT o.OrderNo, o.CustNo, c.Company FROM orders o
LEFT JOIN dbdemos.customer c ON o.CustNo = c.CustNo
WHERE o.CustNo = :OLD_CustNo and OrderNo = :OLD_OrderNo
```

At run time, an application can write a statement directly to this property to set or change the **RefreshRecord** statement.

As the example illustrates, **RefreshRecordSQL** supports an extension to normal parameter binding. To retrieve the value of a field as it exists prior to application of cached updates, the field name with '*OLD_*'. This is especially useful when doing field comparisons in the WHERE clause of the statement.

14.1.7. SQL

Returns a specified SQL statement used when applying cached updates.

Syntax:

```
property SQL[UpdateKind: TUpdateKind]: TStrings;
```

Description:

Returns the SQL statement in the **ModifySQL**, **InsertSQL**, or **DeleteSQL** property, depending on the setting of the **UpdateKind** index. **UpdateKind** can be any of the following:

ukDelete
Return the query object used to execute DELETE statements (**DeleteSQL**).

ukInsert
Return the query object used to execute INSERT statements (**InsertSQL**).

ukModify
Return the query object used to execute UPDATE statements (**ModifySQL**).

14.2. Methods

Please see TMySQLUpdateSQL methods short descriptions below:

Apply
Sets the parameters for a specified SQL statement type, and executes the resulting statement.

Create
Creates an instance of an update object.

Destroy
Frees an instance of an update object.

ExecSQL
Executes a specified type of SQL statement to perform an update for an otherwise read-only results set when cached updates is enabled.

SetParams
Binds parameters in an SQL statement prior to statement execution.

14.2.1. Apply

Sets the parameters for a specified SQL statement type, and executes the resulting statement.

Syntax:

```
procedure Apply(UpdateKind: TUpdateKind);
```

Description:

Call **Apply** to set parameters for an SQL statement and execute it to update a record. **UpdateKind** indicates which SQL statement to bind and execute, and can be one of the following values:

ukDelete
Bind and execute the SQL statement in the **DeleteSQL** property

ukInsert
Bind and execute the SQL statement in the **InsertSQL** property

ukModify
Bind and execute the SQL statement in the **ModifySQL** property

Apply is primarily intended for manually executing UPDATE statements from an **OnUpdateRecord** event handler.

If an SQL statement does not contain parameters, it is more efficient to call **ExecSQL** instead of **Apply**.

14.2.2. Create

Creates an instance of an update object.

Syntax:

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

Description:

Call **Create** to instantiate an update object at run time. You do not need to call **Create** for update objects placed in a data module or form at design time. Delphi automatically handles these objects.

14.2.3. Destroy

Frees an instance of an update object.

Syntax:

```
constructor Destroy;
```

Description:

Do not call **Destroy** directly in an application. Usually destruction of update objects is handled automatically by Delphi. If an application creates its own instance of an update object, however, the application should call **Free**, which verifies that the update object is not **nil** before calling **Destroy**.

14.2.4. ExecSQL

Executes a specified type of SQL statement to perform an update for an otherwise read-only results set when cached updates is enabled.

Syntax:

```
procedure ExecSQL(UpdateKind: TUpdateKind);
```

Description:

Call **ExecSQL** to execute the SQL statement necessary for updating the records belonging to a read-only result set when cached updates is enabled. **UpdateKind** specifies the statement to execute, and can be one of the following values:

ukDelete
Execute the SQL statement used to delete records in the dataset (**DeleteSQL**)

ukInsert
Execute the SQL statement used to insert new records into the dataset (**InsertSQL**)

ukModify
Execute the SQL statement used to update records in the dataset (**ModifySQL**)

If the statement to execute contains any parameters, an application must call SetParams to bind the parameters before calling **ExecSQL**. To determine if a statement contains parameters, examine the appropriate **ModifySQL**, **InsertSQL**, or **DeleteSQL** property, depending on the statement type intended for execution.

To both bind parameters and execute a statement, call **Apply**.

Prepared statements feature is used during the call for performance reason if it's possible.

14.2.5. SetParams

Binds parameters in an SQL statement prior to statement execution.

Syntax:

```
procedure SetParams(UpdateKind: TUpdateKind);
```

Description:

Call **SetParams** to bind parameters in an SQL statement associated with the update object prior to executing the statement. **UpdateKind** indicates the type of statement for which to bind parameters, and can be one of the following values:

ukDelete
Bind parameters for the SQL statement used to delete records (**DeleteSQL**)

ukInsert
Bind parameters for the SQL statement used to insert new records (**InsertSQL**)

ukModify
Bind parameters for SQL statement used to update records (**ModifySQL**)

Parameters are indicated in an SQL statement by a colon. Except for the leading colon in the parameter name, the parameter name must exactly match the name of an existing field name for the dataset.

Parameter names can be prefaced by the *'OLD_'* indicator. If so, the old value of the field is used to perform the update instead of any updates in the cache.

15. FAQ

I've purchased DAC for MySQL, but I keep getting the nag (trial) screen. How can I get rid of it?

I've created new project with C++Builder, put some DAC for MySQL components on the form and run it. I have an Access violation right after start. What can I do?

How can I set database connection properties (eg. coCompress) from code?

What do I need to use SSL-encrypted connections?

I have encountered a performance problem with reloading resultset. What can I do?

Should I use TMySQLUpdateSQL everytime with TMySQLQuery?

How to use Unicode data in my application?

My Delphi 5 application with DAC for MySQL components fails right after start with "Access violation" exception. What shoul I do?

15.1. 1.I've purchased DAC for MySQL, but I keep getting the nag(trial) screen. What can I do?

Q. I've purchased DAC for MySQL, but I keep getting the nag (trial) screen. How can I get rid of it?

A. Please do the following:

1. Close your IDE if it is running
2. Uninstall all installed versions of DAC for MySQL
3. Find and manually delete all MySQLDAC*.bpl and dcl_MySQLDAC*.bpl files, where "*" is your Delphi/C++Builder version
4. Install DAC for MySQL using installer of the full version downloaded from http://microolap.com/my/downloads/.

Questions list

15.2. 2.I've created new project with C++Builder, put some DAC for MySQL components on the form and run it. I have an Access violation right after start. What can I do?

Q. I've created new project with C++Builder, put some DAC for MySQL components on the form and run it. I have an Access violation right after start. What can I do?

A. Uncheck "Build with run-time packages" and "Use dynamic RTL" checkboxes in Project Options dialog, and then rebuild your project.

Questions list

15.3. 3.How can I set database connection properties (eg. coCompress) from code?

Q. How can I set database connection properties (eg. coCompress) from code?

A. Try the following code:

```
mySQLDatabase1.ConnectOptions := mySQLDatabase1.ConnectOptions + [coCompress];
```

Don't forget to add **mySQLTypes** to your uses list.

Questions list

15.4. 4.What do I need to use SSL-encrypted connections?

Q. What do I need to use SSL-encrypted connections?

A. There are a lot of Windows versions of OpenSSL binaries compiled by different people, and some of them may cause problems when using them with DAC for MySQL.

We've tested DAC for MySQL with binaries downloaded from http://www.openssl.org/related/binaries.html.

To make DAC for MySQL use SSL-encrypted connections you should do the following:
1. Setup SSLKey and SSLCert properties of TMySQLDatabase component
2. Put DLLs downloaded from OpenSSL.org near your application EXE file.

Please note:
Your MySQL server must support SSL connections. To check if running MySQL server supports SSL, examine the value of the ***have_openssl*** system variable:

```
mysql> SHOW VARIABLES LIKE 'have_openssl';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| have_openssl  | YES   |
+---------------+-------+
```

If the value is YES, the server supports SSL connections.
If the value is DISABLED, the server supports SSL connections, but it was not started with the appropriate ***--ssl-xxx*** option.
If the value is NO, the server does not support SSL connections.

Questions list

15.5. 5.I have encountered a performance problem with reloading resultset. What can I do?

Q. I have encountered a performance problem. I have a TMySQLQuery (or TMySQLTable) that returns all records in a MySQL table and displays them in a DBGrid. I also have a TMySQLUpdateSQL that is used to write changes back to the table. If I change a field in the table, using the DBGrid, the TMySQLQuery seems to reload the entire dataset again. I would expect that the component would just update itself with the new record, instead of reloading the entire table. For this small table it is not too much of a problem, but for tables with a large number of records it is a significant performance hit. The other components I am evaluating, the entire table is not reloaded after the update.

A. Your expectations are absolutely right for single user local Database Systems (e.g. MS Access), but MySQL is a multiuser server. This means, that each client works not with data, but with a snapshot of data at the moment. This also called transaction schema.
Each user application must have most fresh (actual) snapshot of data, otherwise it may cause warnings by its actions, e.g. UPDATE rows which were deleted or changed by another user etc.
The developer's task is to solve this problem. One of the approaches is to use timeouts to get fresh result set. However, this is not the subject of this answer.

> I would expect that the component would just update itself with the new record, instead of reloading the entire table.

But data may be changed a lot. Moreover, even the record posted to server may be changed by server logic (e.g. by triggers or rules). Data in this record may also affect on other records, for example, in a table which holds tree structure: deleting some parent node will cause deleting of all descendants. That's why we reload whole result set.

> For this small table it is not too much of a problem, but for tables with a large number of records it is a significant performance hit.

Users need a huge result set very seldom. More often they prefer to work with "pages", which may be done by LIMIT and OFFSET clauses of SELECT statement.

> The other components I am evaluating, the entire table is not reloaded after the update.

We believe, that this is an extension of such products, but not standard state, i.e. some properties allow this to be done.
By the way, our TMySQLTable component has BatchModify property that allows such mode to be emulated.

Questions list

15.6. 6.Should I use TMySQLUpdateSQL everytime with TMySQLQuery?

Q. Should I use TMySQLUpdateSQL everytime with TMySQLQuery?

A. No, it should be used only for the following queries types.

Multitable queries:

```
SELECT table1.field1, table2.field2, func.field1 FROM table, table2;
```

Queries with operators in the field list:

```
SELECT field1 + field2, some_function(field3) FROM some_table;
```

Queries with constants or function calls in the field list:

```
SELECT 2, 'char constant', version FROM ...;
```

In other words, Query may be modified automatically (without using TMySQLUpdateSQL), if it consists of plain field list taken from one and only one table.

Questions list

15.7. 7.How can I use Unicode data in my application?

Q. How to use Unicode data in my application?

A. Delphi/C++Builder support Unicode strings starting from 2009 version (Tiburon). DAC for MySQL supports Unicode data starting from v2.7.0. So, to support Unicode in your application

you should use Delphi/C++Builder at least version 2009 and DAC for MySQL at least version 2.7.0.
Another issue for Unicode-enabled DB-applications is connection characterset. DAC for MySQL support Unicode strings only for UTF8 connection characterset. You have to set TMySQLDatabase.ConnectionCharacterSet property to 'utf8' to make DAC for MySQL represent strings as Unicode strings.
Connection character set can be UTF8 even if you don't set TMySQLDatabase.ConnectionCharacterSet property to 'utf8', e.g. if the server is configured to use UTF8 as default connection character set. You can use TMySQLDatabase.Utf8Used property to ensure that your application is connected to MySQL server using UTF8 connection character set.
If TMySQLDatabase.Utf8Used property value is **False** all strings are treated as sinlge-byte strings with ANSI encoding. Such ANSI-behaviuor is the same as for DAC for MySQL before 2.7.0 version.

Questions list

See also: TMySQLDatabase.ConnectionCharacterSet, TMySQLDatabase.Utf8Used properties

15.8. 8. My Delphi 5 application with DAC for MySQL components fails right after start with AV. What should I do?

Q. My Delphi 5 application with DAC for MySQL components fails right after start with "Access violation" exception. What should I do?

A. There are some problems with Delphi 5 code optimizer that cause DAC for MySQL applications failure right after start. There are three possible solutions:

1. Use "Install binaries" option during installation to use precompiled DAC for MySQL units.
2. Turn off code optimization in Project Options dialog.
3. Increase stack size to $00200000 in project options.

Questions list

16. Examples

AfterDelete, Format

Append, FieldValues, Post

BeforeInsert, Insert, AsInteger, FieldByName

BeforePost, Abort

Create, CreateBlobStream, Edit, CopyFrom

CreateTable method usage

DataSetCount, DataSets

DisableControls, EnableControls, Eof

EditKey, GotoKey

EditRangeStart, EditRangeEnd, FieldByName, ApplyRange

EmptyTable

FieldCount, Fields, FieldName

FindField, AsString

FindNearest

GetBookmark, GotoBookmark, FreeBookmark, FindPrior, Value, OnDataChange, BOF

IndexDefs, Update, Count, Items, IndexName, Fields, Name

IndexFields, IndexFieldCount

MasterSource, MasterFields

Min, Max, Position, RecordCount, First, Next

MoveBy, SelectedIndex, Tag

ParamCount, DataType, StrToIntDef, AsXXX

SetKey, GotoNearest

SetRange, CancelRange, Refresh

SQL, ExecSQL

State, Seek, Truncate

16.1. AfterDelete, Format

This example displays a message on the form's status bar indicating the table's record count after a record is deleted.

```
procedure TForm1.MySQLTable1AfterDelete(DataSet: TDataSet);
begin
  StatusBar1.SimpleText := Format('There are now %d records in the table',
                                  [DataSet.RecordCount]);
end;
```

16.2. Append, FieldValues, Post

This example appends a new record to a table when the user clicks a button. The two fields with names ALPHANUMERICFIELD and INTEGERFIELD are filled from the contents of two edit controls.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   MySQLTable1.Append;
   MySQLTable1.FieldValues['ALPHANUMERICFIELD'] := Edit1.text;
   MySQLTable1.FieldValues['INTEGERFIELD'] := StrToInt(Edit2.text);
   MySQLTable1.Post;
end;
```

16.3. BeforeInsert, Insert, AsInteger, FieldByName

This example uses the **BeforeInsert** event to do data validation; if the **StrToInt** function raises an exception, the edit control's contents are set to a valid value so the assignment to the INTEGER field in the table will succeed.

```
procedure TForm1.MySQLTable1BeforeInsert(DataSet: TDataSet);
begin
  try
// Make sure edit field can be converted to integer -
// - otherwise this will raise an exception.
    StrToInt(Edit1.Text);
  except
    Edit1.Text := '0';
  end;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  MySQLTable1.Insert;
  MySQLTable1.FieldByName('QUANTITY').AsInteger := StrToInt(Edit1.Text);
  MySQLTable1.Post;
end;
```

16.4. BeforePost, Abort

This example checks for a valid entry in a **TDBEdit** control and calls the **Abort** procedure if the control is empty; **Abort** cancels the post before it happens.

```
procedure TForm1.MySQLTable1BeforePost(DataSet: TDataSet);
begin
   if DBEdit1.Text = '' then
      Abort;
end;
```

16.5. Create, CreateBlobStream, Edit, CopyFrom

The following example copies the data in the *Notes* field of *MySQLTable1* to the *Remarks* field of *MySQLTable2*.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Stream1, Stream2 : TBlobStream;
begin
  Stream1 := TBlobStream.Create(MySQLTable1Notes, bmRead);
  try
    MySQLTable2.Edit;
// Here's a different way to create a blob stream
    Stream2 := MySQLTable2.CreateBlobStream(MySQLTable2.FieldByName('Remarks'),
                                            bmReadWrite);
    try
      Stream2.CopyFrom(Stream1, Stream1.Size);
      MySQLTable2.Post;
    finally
      Stream2.Free;
    end;
  finally
    Stream1.Free;
  end;
end;
```

16.6. CreateTable() method usage

This example demostrates **TMySQLTable.CreateTable** method used to create database tables at run-time.

```
mySQLTable1.TableName := 'sometable'; //table name
  //integer field
  with mySQLTable1.FieldDefs.AddFieldDef do
  begin
    Name := 'ID';
    DataType := ftInteger;
    Required := True;
  end;
  //string field
  with mySQLTable1.FieldDefs.AddFieldDef do
  begin
    Name := 'VarcharField';
    DataType := ftString;
    Size := 255;
    Required := True;
  end;
  //primary key
  with mySQLTable1.IndexDefs.AddIndexDef do
  begin
    Options := [ixPrimary];
    Fields := 'ID';
  end;
```

```
  //some other index
  with mySQLTable1.IndexDefs.AddIndexDef do
  begin
    Options := [ixUnique];
    Fields := 'VarcharField';
    Name := 'idxForStringField';
  end;
  mySQLTable1.CreateTable;
```

See also: CreateTable method

16.7. DataSetCount, DataSets

The following code fragment illustrates how **DataSets** and **DataSetCount** can be used to ensure that an action is taken for every open dataset that is a table.

After that all open datasets will be closed.

```
var
  I: Integer;
begin
  with MySQLDatabase1 do
  begin
     while DataSetCount <> 0 do
     begin
        if DataSets[0] is TMySQLTable then
        begin
         // Some code
        end;
        DataSets[0].Active := False;
     end;
  end;
end;
```

16.8. DisableControls, EnableControls, Eof

Usually **DisableControls** is called within the context of a *try...finally* block that re-enables the controls even if an exception occurs.

For example:

```
with CustMySQLTable do
begin
  DisableControls;
  try
    First;
    while not Eof do
    begin
     // Process each record here
     Next;
```

```
    end;
  finally
    EnableControls;
  end;
end;
```

16.9. EditKey, GotoKey

The following code uses the **EditKey** and **GotoKey** methods to move to a particular record on *MySQLTable1*.

The actual field values are not changed when making the assignments because of the call to **EditKey**.

```
with MySQLTable1 do
begin
  EditKey;
  FieldByName('State').AsString := 'CA';
  FieldByName('City').AsString := 'Santa Barbara';
  GotoKey;
end;
```

16.10. EditRangeStart, EditRangeEnd, FieldByName, ApplyRange

To get result set where *Company* field value lays between *Edit1* and *Edit2*:

```
with Customer do
begin
  EditRangeStart;
  // Set start range based on text of Edit1 component
  FieldByName('Company').AsString := Edit1.Text;
  EditRangeEnd;
  // Set end range based on value of Edit2 component
  FieldByName('Company').AsString := Edit2.Text;
  ApplyRange; // Apply the ranges
end;
```

16.11. EmptyTable

The following example uses a table component to empty a database table. First the properties of the table component are set to specify the table that should be emptied.

```
with MySQLTable1 do
 begin
  Active := False;
  TableName := 'CustInfo';
  EmptyTable;
 end;
```

16.12. FieldCount, Fields, FieldName

This example displays a message box with the names of all fields in the table.

```
procedure TForm1.Button2Click(Sender: TObject);
var
   i: Integer;
   Info: String;
begin
   Info := 'The fields of table ' + MySQLTable1.TableName +
           ' are:'#13#10#13#10;
   for i := 0 to MySQLTable1.FieldCount - 1 do
      Info := Info + MySQLTable1.Fields[i].FieldName + #13#10;
  ShowMessage(Info);
end;
```

16.13. FindField, AsString

Two ways to modify field value:

```
with MySQLTable1 do
begin
  // This is the safe way to change 'CustNo' field
  FindField('CustNo').AsString := '1234';
  // This is *not* the safe way to change 'CustNo' field
  Fields[0].AsString := '1234';
end;
```

16.14. FindNearest

The following example performs an incremental search on a table.

The form contains a dbgrid, an edit box, a data source, and a table. As the user types in the edit box, the cursor of the grid moves to the nearest match in the table.

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  MySQLTable1.TableName := 'Customer';
  MySQLTable1.Active := True;
  MySQLTable1.IndexName := 'ByCompany';
end;
procedure TForm1.Edit1Change(Sender: TObject);
begin
    MySQLTable1.FindNearest([Edit1.Text]);
end;
```

16.15. GetBookmark, GotoBookmark, FreeBookmark, FindPrior, Value, OnDataChange, BOF

This example uses a button to copy the value of a field in the previous record into the corresponding field in the current record.

```
procedure TForm1.CopyDataClick(Sender: TObject);
var
   SavePlace: TBookmark;
   PrevValue: Variant;
begin
   with MySQLTable1 do
   begin
    // Get a bookmark so that we can return to the same record
    SavePlace := GetBookmark;
    // Move to prior record
    FindPrior;
    // Get the value
    PrevValue := Fields[0].Value;
    // Move back to the bookmark
    // this may not be the next record anymore
    // if something else is changing the dataset asynchronously
    GotoBookmark(SavePlace);
    // Set the value
    Fields[0].Value := PrevValue;
    // Free the bookmark
    FreeBookmark(SavePlace);
  end;
end;
```

To ensure that the button is disabled when there is no previous record, the **OnDataChange** event of the **DataSource** detects when the user moves to the beginning of file (**Bof** property becomes **True**), and disables the button.

```
procedure TForm1.Table1DataChange(Sender: TObject; Field: TField);
begin
  if MySQLTable1.BOF then
    CopyData.Enabled := False
  else
    CopyData.Enabled := True;
end;
```

16.16. IndexDefs, Update, Count, Items, IndexName, Fields, Name

This example uses the **IndexName** property to sort the records in a table on the *CustNo* and *OrderNo* fields.

```
MySQLTable1.Active := False;
// Get the current available indexes
MySQLTable1.IndexDefs.Update;
// Find one which combines Customer Number ('CustNo')
```

```
// and Order Number ('OrderNo')
for I := 0 to MySQLTable1.IndexDefs.Count - 1 do
  if MySQLTable1.IndexDefs.Items[I].Fields = 'CustNo;OrderNo' then
// Set that index as the current index for the table
    MySQLTable1.IndexName := MySQLTable1.IndexDefs.Items[I].Name;
MySQLTable1.Active := True;
```

16.17. IndexFields, IndexFieldCount

The following code calculates the total length of the index and assigns it to the variable **TotalLen**.

```
TotalLen := 0;
with MySQLTable1 do
  for I := 0 to IndexFieldCount - 1 do
    Inc(TotalLen, IndexFields[I].DataSize);
```

16.18. MasterSource, MasterFields

Suppose you have a master table named Customer that contains a *CustNo* field, and you also have a detail table named *Orders* that also has a *CustNo* field.

To display only those records in *Orders* that have the same *CustNo* value as the current record in *Customer*, write this code:

```
Orders.MasterSource := 'CustSource';
Orders.MasterFields := 'CustNo';
```

If you want to display only the records in the detail table that match more than one field value in the master table, specify each field and separate them with a semicolon.

```
Orders.MasterFields := 'CustNo;SaleDate';
```

16.19. Min, Max, Position, RecordCount, First, Next

To read through all records in a table and update the **ProgressBar** accordingly:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
begin
  with ProgressBar1 do
  begin
    Min := 0;
    Max := MySQLTable1.RecordCount;
    MySQLTable1.First;
    for i := Min to Max do
    begin
      Position := i;
      MySQLTable1.Next;
```

```
    end;
  end;
end;
```

16.20. MoveBy, SelectedIndex, Tag

The following example enables the user to move the current selected cell in a dbgrid. The **Up** and **Down** buttons have their **OnClick** events assigned to the **UpDownClick** procedure. The **Left** and **Right** buttons have their **OnClick** events assigned to the **LeftRightClick** procedure. The **Up** and **Left** buttons have their **Tag** property set to **-1**, while the **Down** and **Right** buttons have their **Tag** property set to **1**.

```
procedure TForm1.UpDownClick(Sender: TObject);
begin
  MySQLTable1.MoveBy(TComponent(Sender).Tag);
  DBGrid1.SetFocus;
end;
procedure TForm1.LeftRightClick(Sender: TObject);
begin
  DBGrid1.SelectedIndex := DBGrid1.SelectedIndex + TComponent(Sender).Tag;
  DBGrid1.SetFocus;
end;
```

16.21. ParamCount, DataType, StrToIntDef, AsXXX

This example fills in the parameters of a query from the entries of a list box.

```
var
  I: Integer;
  ListItem: String;
begin
  for I := 0 to MySQLQuery1.ParamCount - 1 do
  begin
    ListItem := ListBox1.Items[I];
    case MySQLQuery1.Params[I].DataType of
      ftString:
        MySQLQuery1.Params[I].AsString := ListItem;
      ftSmallInt:
        MySQLQuery1.Params[I].AsSmallInt := StrToIntDef(ListItem,0);
      ftInteger:
        MySQLQuery1.Params[I].AsInteger := StrToIntDef(ListItem,0);
      ftWord:
        MySQLQuery1.Params[I].AsWord := StrToIntDef(ListItem,0);
      ftBoolean:
        begin
          if ListItem = 'True' then
            MySQLQuery1.Params[I].AsBoolean := True else
            MySQLQuery1.Params[I].AsBoolean := False;
        end;
      ftFloat:
        MySQLQuery1.Params[I].AsFloat := StrToFloat(ListItem);
      ftCurrency:
        MySQLQuery1.Params[I].AsCurrency := StrToFloat(ListItem);
```

```
      ftBCD:
        MySQLQuery1.Params[I].AsBCD := StrToCurr(ListItem);
      ftDate:
        MySQLQuery1.Params[I].AsDate := StrToDate(ListItem);
      ftTime:
        MySQLQuery1.Params[I].AsTime := StrToTime(ListItem);
      ftDateTime:
        MySQLQuery1.Params[I].AsDateTime := StrToDateTime(ListItem);
    end;
  end;
end;
```

16.22. Prepared, Prepare

```
if not MySQLQuery1.Prepared then
begin
  MySQLQuery1.Close;
  MySQLQuery1.Prepare;
  MySQLQuery1.Open
end;
```

16.23. SetKey, GotoNearest

To set cursor to row in which *City* field begins with *'Santa'*:

```
with MySQLTable1 do
begin
  SetKey;
  FieldByName('State').AsString := 'CA';
  FieldByName('City').AsString  := 'Santa';
  GotoNearest;
end;
```

See also: TMySQLTable.SetKey, TMySQLTable.GotoNearest methods

16.24. SetRange, CancelRange, Refresh

The following example sets a range for a table. The form requires two edit boxes, a data source, a table, a dbgrid, and a button.

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  MySQLTable1.TableName := 'Customer';
  MySQLTable1.Active    := True;
  MySQLTable1.IndexName := 'ByCompany';
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
```

```
  if Button1.Caption = '&Apply Range' then
    begin
      MySQLTable1.SetRange([Edit1.Text],[Edit2.Text]);
      Button1.Caption := '&Drop Range';
    end
  else
    begin
      MySQLTable1.CancelRange;
      MySQLTable1.Refresh;
      Button1.Caption := '&Apply Range';
    end;
end;
```

16.25. SQL, ExecSQL

To execute SQL statement deleting all rows where *Country* field value is equal to *Argentina*:

```
MySQLQuery1.Close;
MySQLQuery1.SQL.Clear;
MySQLQuery1.SQL.Add(
  'Delete from Country where Name = ''Argentina''');
MySQLQuery1.ExecSQL;
```

16.26. State, Seek, Truncate

The following example deletes the stream from position 60 within the blob stream to the end.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Stream1: TBlobStream;
begin
  MySQLTable1.Edit;
  if MySQLTable1.State = dsEdit then
  begin
    Stream1 := MySQLTable1.CreateBlobStream(FieldByName('Notes', bmReadWrite);
    try
      Stream1.Seek(60, 0); // Move to byte 60
      Stream1.Truncate;    // Delete from current position (60) to end of stream.
      MySQLTable1.Post;
    finally
      Stream1.Free;
    end;
  end;
end;
```

17. DataTypes map

This section shows how various MySQL datatypes are mapped to Borland/CodeGear/Embarcadero's **TField** descendants.

MySQL datatypes	TField descedant
BIT, BIGINT	TLargeintField (see warning below)
TINYINT(1), BOOL, BOOLEAN	TSmallIntField or TBooleanField (see note #4 below)
other TINYINT	TSmallIntField
SMALLINT	TSmallIntField
MEDIUMINT, INT, INTEGER, YEAR	TIntegerField
UNSIGNED INT	TLargeintField
FLOAT, DOUBLE, DECIMAL, DEC, FIXED	TFloatField
DATE	TDateField
DATETIME, TIMESTAMP	TDateTimeField
TIME	TTimeField
CHAR	TStringField (or TWideStringField, see note #1 below)
VARCHAR(<8192)	TStringField (or TWideStringField, see notes #1 and #2 below)
VARCHAR(>=8192)	TMemoField (or TWideMemoField, see notes #1 and #2 below)
BINARY, VARBINARY, TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB	TBlobField
TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT	TMemoField (or TWideMemoField, see notes #1 and #5 below)

MySQL datatypes	TField descedant
ENUM('n','y'), ENUM('f','t')	TBooleanField or TStringField (see note #4 below)
other ENUM	TStringField (or TWideStringField, see note #1 below)
SET	TStringField (or TWideStringField, see note #1 below)

#1
TStringField and **TMemoField** can be replaced with **TWideStringField** and **TWideMemoField** respectively for Unicode data in Delphi/C++Builder 2009. Please read this FAQ section if you want to use Unicode strings in your application: How can I use Unicode data in my application?

#2
Be careful while working with Unicode encoded fields (e.g., utf8) in "pre-Delphi-2009" projects. These fields length is calculated in bytes, not in symbols. So varchar(3000) can contain from 3000 to 12000 bytes. This means that it can be mapped to **TStringField** or to **TMemoField** in different cases for "pre-Delphi-2009" projects.

#3
If you want to use TNT Unicode Controls to implement Unicode support to your "pre-Delphi-2009" application you can use our free package with wrapper-components - Wrappers for TNT Unicode Controls. All string fields are mapped to **TTNTStringField** datatype with this package.

#4
DAC for MySQL treats *ENUM('n','y')* or *ENUM('f','t')* MySQL datatypes as **TBooleanField** fields and TINYINT datatype as **TSmallIntField** by default. But MySQL Reference Manual says that BOOL and BOOLEAN datatypes are synonyms for *TINYINT(1)* and even defines **True** (as **1**) and **False** (as **0**) constants for it. You can let DAC for MySQL to treat BOOL, BOOLEAN and *TINYINT(1)* datatypes as **TBooleanField** by enabling M_BOOL_AS_INT conditional define in *mySQLDAC.inc* file and rebuilding DAC for MySQL packages. All ENUM fields are mapped to **TStringField** (or **TWideStringField**) fields in this case.

#5
If you use 'utf8' character set and *'utf8_bin'* collation for TINYTEXT, TEXT, MEDIUMTEXT or LONGTEXT column MySQL marks it with BINARY flag. This cause this columns to be mapped by DAC for MySQL to **TBlobField** rather then to **TMemoField**. This can break correct Unicode data handling in "Delphi-2009" projects. Please consider use 'utf8_general_ci' or *'utf8_unicode_ci'* collation for MEMO columns for proper Unicode texts handling.

Delphi 7 and prior has poor support for *int64* values in *variant* type. This means that you'll be unable to use Locate and similar methods with such fields. Lookup fields and master-detail tables will not work properly too because of their dependence on **Locate** method. Please update your IDE to BDS 2006 (or later) or don't use BIGINT and UNSIGNED INT datatypes for keys and indexes if you need to use them in lookup fields. Also you can't use variant-style properties (**FieldValues**, **AsVariant** and so on) with such fields.

18. License Agreement

NOTICE TO USER:

THIS IS AN AGREEMENT GOVERNING YOUR USE OF THE SOFTWARE TITLED **Microolap DAC for MySQL**, FURTHER DEFINED HEREIN AS "PRODUCT," AND THE LICENSOR OF THE PRODUCT IS WILLING TO PROVIDE YOU WITH ACCESS TO THE PRODUCT ONLY ON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS AND CONDITIONS CONTAINED IN THIS AGREEMENT. BELOW, YOU ARE ASKED TO ACCEPT THIS AGREEMENT AND CONTINUE TO INSTALL OR, IF YOU DO NOT WISH TO ACCEPT THIS AGREEMENT, TO DECLINE THIS AGREEMENT, IN WHICH CASE YOU WILL NOT BE ABLE TO INSTALL OR OPERATE THE PRODUCT. BY ACKNOWLEDGING YOUR CONSENT HERETO AND BY INSTALLING AND OPERATING THIS PRODUCT YOU ACCEPT ALL THE TERMS AND CONDITIONS OF THIS AGREEMENT. For purposes hereof ***"Operating"*** shall mean accessing, storing, loading, installing, Using (as defined below), and copying the Product into the memory of a Client Device, as defined below. ***"Using"*** shall mean executing and displaying the product on a Client Device or otherwise benefiting from utilizing, deploying or using the Product or its functionality.

This Electronic End User License Agreement (the ***"Agreement"***) is a legal agreement between you (either an individual or an entity), the Licensee, and MicroOLAP Technologies Ltd., (the ***"Licensor"***), regarding the Product and related support service you are about to install and Operate and/or other related services, including without limitation:

a) all of the contents of the files, including disk(s), CD-ROM(s) or other media with which this Agreement is provided and including all forms of code, such as Source Code, Object Code, dynamic or static libraries, and/or executable files as provided and in a form that is provided by Licensor to you (the ***"Software"***). For the avoidance of doubt, by way of example, but not exclusion, if a specific file is provided by Licensor in Object Code only, the Source Code for such files shall not be deemed a part of the Software provided by Licensor to you. For purposes hereof ***"Source Code"*** shall mean the human-readable form of the computer programming code and related system documentation including all comments and any procedural code such as job control language and ***"Object Code"*** shall mean computer programs assembled or compiled in magnetic or electronic binary form on software media, which are readable and usable by machines, but not generally readable by humans without reverse-assembly, reverse-compiling, or reverse-engineering.

b) all support services provided to you by Licensor in connection with the Software (the ***"Services"***);

c) and all successor upgrades, modified versions, modified modules, revisions, patches, enhancements, fixes, modifications, copies, additions or maintenance releases of the Software, if any, licensed to you by the Licensor (collectively, the ***"Updates"***), and

d) related user documentation and explanatory materials or files provided in written, "online" or electronic form (the ***"Documentation"*** and together with the Software, Samples, Updates, and Services the ***"Product"***).

For the purposes of this Agreement, ***"Licensor Site"*** shall mean the Internet website maintained by or on behalf of Licensor from which the Software is available for download pursuant to a license from Licensor. The Licensor Site is currently located at ***http://www.microolap.com***.

You are subject to the terms and conditions of this End User License Agreement whether you access or obtain the Product directly from the Licensor, or through any other source. For purposes hereof, ***"you"*** or ***"Licensee"*** means the individual person installing or using the Product on his or her own behalf; or, if the Product is being downloaded or installed on behalf of an organization, such as an employer, ***"you"*** means the organization for which the Product is downloaded or installed, then the person accepting this agreement represents hereby that such organization has authorized such person to accept this agreement on the organization's behalf. For purposes hereof the term ***"organization,"*** without limitation, includes any partnership, limited liability company, corporation, association, joint stock company, trust, joint venture, labor organization, unincorporated organization, or governmental authority.

If you do not agree to the terms and conditions of this Agreement, the Licensor is unwilling to license the Product to you. In such event, you may not Operate the Product in any way.

IF THE LICENSOR AND YOU HAVE AGREED ON AND PROPERLY EXECUTED A SEPARATE CONTEMPORANEOUS OR SUBSEQUENT TERMS OF USE OR EXHIBITS (the *"Terms of Use"*), WHICH ARE SUPPLEMENTAL, DIFFERENT OR INCONSISTENT WITH THE TERMS OF THIS AGREEMENT, SUCH TERMS OF USE SHALL CONTROL, PROVIDED THAT (i) SUCH TERMS OF USE SPECIFICALLY ACKNOWLEDGE AND REFER TO THIS AGREEMENT, AND (ii) ALL OTHER TERMS AND CONDITIONS OF THIS AGREEMENT REMAIN IN FULL FORCE AND EFFECT.

BEFORE YOU PUT A CHECKMARK AT THE "I ACCEPT THE AGREEMENT" BUTTON AND PRESS "NEXT," PLEASE CAREFULLY READ THE TERMS AND CONDITIONS OF THIS AGREEMENT, AS SUCH ACTIONS ARE A SYMBOL OF YOUR SIGNATURE AND BY CLICKING ON THE "I ACCEPT THE AGREEMENT" AND "NEXT" BUTTONS, YOU ARE CONSENTING TO BE BOUND BY AND ARE BECOMING A PARTY TO THIS AGREEMENT AND AGREE THAT THIS AGREEMENT IS ENFORCEABLE LIKE ANY WRITTEN NEGOTIATED AGREEMENT SIGNED BY YOU. IF YOU DO NOT AGREE TO ALL OF THE TERMS OF THIS AGREEMENT, CLICK THE "CANCEL" BUTTON AND THE PRODUCT WILL NOT BE INSTALLED ON YOUR CLIENT DEVICE, AS SUCH TERM IS DEFINED BELOW.

For your reference, you may refer to the copy of this Agreement that can be found in installed files of the Software as **license.rtf**.

You may also receive an electronic copy of this Agreement by contacting Licensor at ***sales@microolap.com***.

1. Proprietary Rights and Non-Disclosure.

1.1. Ownership Rights. You agree that the Product and the authorship, systems, ideas, methods of operation, derivative Documentation and other information contained in the Product, are proprietary intellectual properties and or the valuable trade secrets of the Licensor and are protected by applicable civil and criminal law, and by the law of copyright, trade secret, trademark and patent and international treaties. You may use trademarks only insofar as to identify printed output produced by the Product in accordance with accepted trademark practice, including identification of trademark owner's name. Such use of any trademark does not give you any rights of ownership in that trademark. The Licensor and its suppliers own and retain all right, title, and interest in and to the Product, including all copyrights, patents, trade secret rights, trademarks, and other intellectual property rights therein. Your possession, installation or use of the Product does not transfer to you any title to the intellectual property in the Product, and you will not acquire any rights to the Product except as expressly set forth in this Agreement. All copies of the Product made hereunder must contain the same proprietary notices that appear on and in the Product. Except as stated herein, this Agreement does not grant you any intellectual property rights in the Product.

1.2. Source Code and Modifications. You acknowledge that the source code for the Product is proprietary to the Licensor and constitutes trade secrets of the Licensor. Except as otherwise specifically provided herein or in Terms of Use, you agree not to disassemble, decompile or "unlock", decode or otherwise reverse-translate or reverse-engineer, or attempt in any manner to reconstruct or discover any source code or underlying algorithms of the Product or any part thereof provided solely in Object Code form but you may change, add or delete any files of the licensed copy of the Products and you may adapt or modify the Source Code solely for purposes of Operating a licensed copy of the Product and as expressly permitted pursuant to the type of the License purchased hereunder *provided* that you may not, in any event, remove or alter any copyright notices or other proprietary notices on any copies of the Product, whether so modified or not, *and further provided* that any such change, addition, deletion, adaptation or modification voids any express warranty provided herein and terminates any right to support services.

1.3. Confidential Information. You agree that, unless otherwise specifically provided herein the Product, including the specific design and structure of individual programs and the Product, constitute confidential proprietary information of the Licensor or its suppliers and/or licensors. You agree not to transfer, copy, disclose, provide or otherwise make available such confidential information in any form to any third party. For purposes hereof, ***"License Key"*** or ***"Registration Key"*** shall mean a file or a unique sequence of digit and/or symbols provided to you by the Licensor confirming the purchase of the license from the Licensor, which may carry the information about the License, i.e. its type, the user name and the number of licenses purchased, and enabling the full functionality of the Product in accordance with the License granted under this Agreement. You agree to implement reasonable security measures to protect such confidential information. If you download the Software from the Internet or similar on-line source, you must include the copyright notices resident on the Software with any on-line distribution and on any

media you distribute that includes the Software.

2. **Grant of License.**

2.1. License. Unless otherwise specifically indicated under a valid Terms of Use, the Licensor grants you a non-exclusive and non-transferable license without the right to sublicense (the ***"License"***) and Licensee hereby accepts such License as follows, *provided* that unless otherwise agreed by Licensor or specified by Terms of Use, each License is granted per one Licensee:

a) Trial License. If you have received, downloaded and/or installed a Trial Version of the Product (the "Trial Version") and are hereby granted a Trial License for the Software and you may Operate the Product only for evaluation and demonstration purposes and only during the single applicable evaluation period of thirty (30) days (the ***"Trial Period"***), unless otherwise indicated, from the date of the initial installation. Any use of the Product for other purposes or beyond the applicable evaluation period is strictly prohibited, *provided however* that, subject to the restrictions contained herein, you may copy and distribute the Trial Version of the Software without any modifications whatsoever, and including this Agreement, to any third party. Licensee shall have no technical support rights during the Trial Period. The Licensor shall not be required to provide any support and Updates, as stated below, for the Trial Version of the Product. During the Trial Period, the Licensor provides no warranty whatsoever and assumes and bears no liability whatsoever for the Trial Version of the Product.

b) Personal Use License. If the Product is licensed under Personal License with Software provided in Object Code only upon the terms specified in the applicable registration, Terms of Use, invoicing or packaging for the Product, you personally may use of the Product solely for Personal Use (***"Personal License"***). ***"Personal Use"*** shall mean personal non-commercial, non-business, non-government Use, and not on behalf or for the benefit of any clients and excludes any commercial purposes whatsoever, which include without limitation: advertising marketing and promotional materials/services on behalf of an actual client, employer, employee or for your own benefit, any products that are commercially distributed, whether or not for a fee, any materials or services for sale or for which fees or charges are paid or received. Upon payment for the License and registration of the Product, you are granted a non-exclusive and non-transferable personal License to (i) install one (1) copy of the Product on up to three (3) Client Devices owned by you, and (ii) subject to the payment of the applicable fees and your compliance with the terms hereof, to Use one (1) copy of the specified version of the Product during the Term of this Agreement, on one (1) of such Client Devices at any given time. Additionally, the individual licensing terms may specify other terms, conditions and restrictions of Operating the Product.

c) Site License. If the Product is licensed with site license terms specified in the applicable product invoicing or packaging for the Product, you may Operate and Use the Product on an unlimited number of Client Devices within a single building owned or leased by your company. Additionally, the individual licensing terms may specify other terms, conditions and restrictions of Using the Product (the ***"Site License"***).

d) Business License. If the Product is licensed under Business License with the Software provided in Object Code only upon the terms specified in the applicable Terms of Use, invoicing or packaging for the Product, you may Use of the Product solely for Business Use (the ***"Business License"***). For purposes hereof, ***"Business Use"*** shall mean business, commercial, government Use only for internal business purposes of such entity without the right to distribute the applications, frameworks or components developed Using the Software (the ***"Results"***) to third parties *provided that* nothing herein shall be construed as creating any obligations of the Licensor to any end user of the Results. Under the Business License, any Use or Operating of the Product or transfer of the results of Using of the Product on a computer device owned by a third party is strictly prohibited unless a separate License is purchased therefor.

e) Commercial License. If the Product is licensed under Commercial License with the Software provided in Object Code only upon the terms specified in the applicable Terms of Use, invoicing or packaging for the Product, you may Use of the Product solely for Commercial Use (the ***"Commercial License"***). For purposes hereof, ***"Commercial Use"*** shall mean business, commercial, government Use with the right to distribute the Results to third parties. Licensee may Use the Product licensed under the Commercial License on an unlimited number of web servers and domains owned, rented or leased by Licensee. Under the Commercial License,

any Use or Operating of the Product or transfer of the results of Using of the Product on a computer device owned by a third party is strictly prohibited unless a separate License is purchased therefor.

f) Intranet License. If the Product is licensed with Intranet License terms specified in the applicable product invoicing or packaging for the Product, you may Operate and Use the Product on an unlimited number of Client Devices within a single local area network and/or private computer network under your control (the ***"Intranet License***"). Additionally, the individual licensing terms may specify other terms, conditions and restrictions of Using the Product.

g) Internet License. If the Product is licensed with Internet License terms specified in the applicable product invoicing or packaging for the Product, you may Operate and Use the Product on an unlimited number of Client Devices on a single server accessible via internet (the ***"Internet License"***). Additionally, the individual licensing terms may specify other terms, conditions and restrictions of Using the Product.

h) Compiled Units. If you are granted a Commercial License pursuant to Section 2.1(e) hereof, in addition to the licenses and rights granted therein, Licensor grants you a nonexclusive, deployment-free, royalty-free right to reproduce and distribute the Object Code version of those portions of the Software which are identified in the Documentation as *'compiled units'* (the ***"Compiled Units"***) *provided* that you comply with all of the following requirements:

i). you distribute the Compiled Units in Object Code form only in conjunction with and as part of your software application product which adds significant and primary functionality and when the absence of Compiled Units will make your software application inoperable;

ii). you do not use Licensor name, logo or trademarks to market your software application product; and

iii). you include a valid copyright, trademark or any other proprietary notices on your Software identifying the Licensor as the owner of the Compiled Units.

i) Source Code License. The Licensor, pursuant to Terms of Use or invoicing terms, and in conjunction with one of Licenses granted under Section 2 hereof, may grant you certain rights to Software provided in Source Code as follows (the ***"Source Code License"***):

i). For purposes hereof, "Source Code" shall mean the human-readable form of the computer programming code and related system documentation including all comments and any procedural code such as job control language. Provided you have purchased a license to a part of the Software supplied in Source Code form, you may make modifications, enhancements, derivative works and/or extensions to that licensed Source Code provided to you under the terms set forth in this Section 2.1(f).

ii). While the Licensor does not claim any ownership rights in the Results, in the event you develop any modifications, enhancements, derivative works and/or extensions to the licensed Source Code (the ***"Derivatives"***), either independently or jointly with the Licensor, such Derivatives and all rights associated therewith will be the exclusive property of the Licensor.

iii). You shall not grant, either expressly or by implication, any rights, title, interest, or licenses to any Derivatives to any third party. You will, however, be entitled to use such Derivatives under the terms set forth in this Agreement. You hereby assign all right, title and interest in and to such Derivatives to the licensed Source Code to the Licensor.

iv). You also agree to execute, acknowledge and deliver to the Licensor all documents and instruments and do all things and actions Licensor deems necessary or desirable, at no cost to you but at Licensor's expense, to enable the Licensor to obtain and secure such Derivatives anywhere in the World. You agree to secure all necessary rights and obligations from relevant employees, or third parties in order to satisfy the above obligations. You may not distribute the Licensor's Source Code, or any Derivatives, in Source Code form.

v). Under no circumstances may any portion of the Source Code be distributed, disclosed or otherwise made available to any third party without the express, prior written consent of the Licensor. Under no circumstances may the Source Code be used in whole or in part,

as the basis for creating a product that provides the same, or substantially the same, functionality as any of the Licensor's product. You will not take any action, or assist or otherwise aid anyone else in taking any action that would, in any way, limit the Licensor's independent development, sale, assignment, licensing or use of its Software or any Derivatives thereof. You will not modify or delete, in whole or part, any copyright, trade secret, proprietary, confidential or other notice thereon or therein, including a prominent notice on the Results "***Powered by Microolap DAC for MySQL***" without the express, prior written consent of the Licensor.

vi). YOU UNDERSTAND, ACKNOWLEDGE AND AGREE THAT SOURCE CODE IS LICENSED "AS IS," AND THAT THE LICENSOR DOES NOT PROVIDE ANY TECHNICAL SUPPORT FOR SOURCE CODE.

j) Business License with Source Code License. If the Product is licensed under Business License with Source Code License upon the terms specified in the applicable Terms of Use, invoicing or packaging for the Product, your rights to Use the Product shall be the rights granted under the Business License together with the rights granted under the Source Code License (the ***"Business License with Source Code License"***).

k) Commercial License with Source Code License. If the Product is licensed under Commercial License with Source Code License upon the terms specified in the applicable Terms of Use, invoicing or packaging for the Product, your rights to may Use of the Product shall be the rights granted under the Commercial License together with the rights granted under the Source Code License (the ***"Commercial License with Source Code License"***).

l) Customized Licenses. The Licensor may grant you a specific customized terms of License pursuant to a valid Terms of Use signed by both parties hereto in which case such the terms and conditions of the Terms of Use shall be controlling and supersede any conflicting terms of this Agreement.

m) Educational Purpose License, Educational Institution Classroom License, and Educational Institution Site License. If the Product is licensed under an Educational Purpose License upon the terms specified in the applicable Educational Purpose License invoicing or packaging for the Product, you may make use of the Product solely for Educational Purpose. ***"Educational Purpose"*** means any non-commercial study or research that is undertaken solely in furtherance of one's education, whether or not completed by a student in pursuit of an educational degree, certificate or diploma and as used by teachers or facilitates teaching of a class, and all administrative staff, faculty and employees, of any college, university, trade school or other school (***"Educational Institution"***). With the acquisition of an Educational Institution Classroom License, Licensee may install and Operate the Product by a number of Users determined by the applicable invoicing terms within one Educational Institution in one classroom. Within these limitations, you may install the Product as a "Network" Product and run the Product from any networked Client Devices on Licensee's LAN, provided that such Client Devices are located exclusively within one classroom. With the acquisition of an Educational Institution Site License, Licensee may install and Operate the Product by a number of Users determined by the applicable invoicing terms within one Educational Institution in one geographic location. Within these limitations, you may install the Product as a "Network" Product and run the Product from any networked Client Devices on Licensee's LAN, provided that such Client Devices are located exclusively within one office complex within one geographic location. Educational License may be granted exclusively at the discretion of the Licensor upon your submission of a written request discussing your and your employer/employees activities, when applicable, and your reasons for and purposes of Operating the Product.

2.2. Multiple Environment Product; Multiple Language Product; Dual Media Product; Multiple Copies; Bundles. If you use different versions of the Product or different language editions of the Product, if you receive the Product on multiple media, if you otherwise receive multiple copies of the Product, or if you received the Product bundled with other software, the total permitted number of your Client Devices on which all versions of the Product are installed shall correspond to the number of licenses you have obtained from the Licensor provided that unless the licensing terms and the License Key provides otherwise, each purchased license entitles you to install and Use the Product on one (1) Client Device. You may not rent, lease, sublicense, lend or transfer any versions or copies of the Product regardless of whether you use the Product or not, *provided that* the terms specified in the applicable product invoicing or packaging for the Product specify otherwise.

2.3. Run-time License. If the Product is licensed under Commercial License in accordance to Sections 2.1(e) or 2.1(k), in addition to the licenses and rights granted therein, Licensor grants Licensee a right to display, install, copy and distribute a portion of Licensor Software (the ***"Run-time Edition"***) on Client Devices of its customers as part of the Licensee's bundled products (the ***"Run-time License"***) in accordance with terms and conditions specified in the License Key, and/or invoicing terms, including the limitations relating to the number Client Devices of such customers or the number of such customers as the case may be, *provided that*, Licensee shall include a valid and prominent copyright, trademark or any other proprietary notices in its bundled products identifying the Licensor as the owner of the Run-time Edition as may be specified by Licensor from time to time.

2.4. Updates. During the Term of this Agreement, you may download Updates to the Product when and as the Licensor publishes them in its website or through other online services. Notwithstanding any provision to the contrary herein, nothing in this Agreement shall be construed as to grant you any rights or licenses with regard to the New Releases of the Product or to entitle you to any New Release. This Agreement does not obligate the Licensor to provide any Updates. Notwithstanding the foregoing, any Updates that you may receive become part of the Product and the terms of this Agreement apply to them (unless this Agreement is superceded by a further Agreement accompanying such Update or modified version of to the Product).

2.5. Material Terms and Conditions. You specifically agree that each of the terms and conditions of this Section 2 are material and that failure of you to comply with these terms and conditions shall constitute sufficient cause for Licensor to immediately terminate this Agreement and the License granted under this Agreement. The presence of this Section 2.5 shall not be relevant in determining the materiality of any other provision or breach of this Agreement by either party hereto.

3. Additional Covenants; Assignment of Intellectual Property Rights.

3.1. Additional Limitations. Notwithstanding anything to the contrary herein, you may not Operate, Use, or modify the Product in any way as to form the basis for creating a product that provides the same, or substantially the same, functionality as the Product; and in the event you develop any modifications, enhancements, derivative works and/or extensions to the Product, either independently or jointly with Licensor, such modifications, enhancements, derivative works and/or extensions and all rights associated therewith will be the exclusive property of Licensor. You will not grant, either expressly or impliedly, any rights, title, interest, or licenses to any such modifications, enhancements, derivative works and/or extensions to any third party. You will, however, be entitled to use such modifications, enhancements, derivative works and/or extensions under the terms set forth in this Agreement. You hereby assign all right, title and interest in and to such modifications, enhancements, derivative works and/or extensions to the Product to Licensor. You also agree to execute, acknowledge and deliver to Licensor all documents and do all things Licensor deems necessary or desirable, at no cost to but at Licensor expense, to enable Licensor to obtain and secure such modifications, enhancements, derivative works and/or extensions anywhere in the world. You agree to secure all necessary rights and obligations from relevant employees or third parties in order to satisfy the above obligations.

3.2. Reservations of all Rights. The Licensor reserves all rights not expressly granted herein.

3.3. Back-up Copies. You can make one (1) copy of the Product for backup and archival purposes, *provided, however*, that the original and each copy is kept in your possession or control, and that your installation, Operation and Use of the Product does not exceed that which is allowed in Section 2 hereof.

3.4. Additional Protection Measures. Solely for the purpose of preventing unlicensed and/or unauthorized Use of the Product, including without limitation Run-time Edition, the Software may collect certain non-personal information relating to the hardware of your or your customer's Client Devices and/or install on your or your customer's Client Devices certain technological measures that are designed to prevent unlicensed and/or unauthorized Use and Operation of the Product, and the Licensor may use this technology to confirm that you have a licensed copy of the Product. Such installation or collection of information or updates of these technological measures may occur through and/or during the installation or activation of the Product or Updates. The Product and/or Updates will not install or may fail to Operate if installed contrary to the rights granted under the License or if attempted to be installed or Operated on unlicensed copies of the Product. If you are not using a licensed copy of the Product, you are not allowed to install the Updates. The Licensor

will not collect any personally identifiable information from your computer during this process.

4. Term and Termination.

4.1. Term. The term of this Agreement (***"Term"***) shall begin when you download, access or install the Product, whichever is earlier, and shall continue in perpetuity unless otherwise designated in the purchase order, Terms of Use, exhibit or unless otherwise terminated pursuant hereto. Without prejudice to any other rights, this Agreement will terminate automatically if you fail to comply with any of the limitations or other requirements described herein. Upon any termination or expiration of this Agreement, you must immediately cease Operating the Product and all of its components and destroy, uninstall and erase all copies of the Product and all of its components, including without limitation on all systems and all types of media and in computer memory.

4.2. No Rights upon Termination. Upon termination of this Agreement you will no longer be authorized to Operate or Use the Product in any way.

5. Support and Updates.

5.1. Terms of Support. During the Warranty Period as defined below, you are entitled to technical services and support for the Product which is provided to you by Licensor free of charge during the regular business hours (GMT+3), except for locally-observed holidays, and includes the support provided through a special technical support section of the Licensor Site (http://www.microolap.com/support/) (***"Warranty Support"***). During the Warranty Period, Warranty Support is unlimited and includes technical and support questions and patch fixes. After the expiration of the Warranty Period, you may purchase additional support services from Licensor at current rates as listed at Licensor Site (http://www.microolap.com/support/).

5.2. Updates. During the Warranty Period hereunder and, if your purchasing terms provide for a certain extended period of time during which you are entitled to Updates free of charge (i.e., one year from the time of the purchase of the License), then for such extended period of time, you may download Updates to the Product when and as the Licensor publishes them on the Licensor Site, or through other online services. If the Product is an Update to a previous version of the Product, you must possess a valid license to such previous version in order to use the Update. You may continue to use the previous version of the Product on your Client Device after you receive the Update to assist you in the transition to the Update, provided that: (i) the Update and the previous version are installed on the same computer device; (ii) the previous version or copies thereof are not transferred to another party unless all copies of the Update are also transferred to such party; and (iii) you acknowledge that any obligation the Licensor may have to support the previous version of the Product may be ended upon availability of the Update. Except for the rights to free Updates during the Warranty Period, as further defined herein, nothing in this Agreement shall be construed as to grant you any rights or licenses with regard to the new version or releases of the Product or to entitle you to any new version, upgrade or release. This Agreement does not obligate the Licensor to provide any Updates. Notwithstanding the foregoing, any Updates that you may receive become part of the Product and the terms of this Agreement apply to them (unless this Agreement is superseded by a succeeding agreement accompanying such Update or modified version of the Product). Notwithstanding anything to the contrary herein, nothing herein shall be deemed to entitle you to any Licensor's Optional Updates that are provided by Licensor for a fee, or to any new releases, versions or substitutes of the Product.

5.3. Additional Support and Updates. In addition to the free Support and free Updates provided for in Sections 5.1 and 5.2, you may purchase additional Services, additional Updates or additional Support beyond the applicable period at the ongoing rates and prices published or provided by Licensor.

6. Restrictions.

6.1. No Transfer of Rights. Except as otherwise specifically provided herein or in the applicable Terms of Use, you may not transfer any rights pursuant to this Agreement nor rent, sublicense, lease, loan or resell the Product or permanently or temporarily transfer the Product in any other manner. You may not permit third parties, including any subcontractors, to benefit from the use or functionality of the Product including, without limitations, via a timesharing, service bureau or other arrangement, except to the extent such use is specified in the application price list, purchase order or product packaging for the Product. Except as otherwise provided in Section 1.2 hereof, you may not, without the Licensor's prior written consent, reverse engineer, decompile,

disassemble or otherwise reduce any part of the Product to human readable form nor permit any third party to do so, except to the extent the foregoing restriction is expressly prohibited by applicable law. Notwithstanding the foregoing sentence, decompiling the Software is permitted to the extent the laws of your jurisdiction expressly give you the non-waivable right to do so to obtain information necessary to render the Software interoperable with other software; provided, however, that you must first request such information from the Licensor and the Licensor may, in its discretion, either provide such information to you (subject to confidentiality terms) or impose reasonable conditions, including a reasonable fee, on such use of the Software to ensure that the Licensor's and its affiliates' proprietary rights in the Software are protected. Except for the modification permitted under Section 1.2, you may not modify, or create derivative works based upon the Product in whole or in part.

6.2. No Extraction for Separate Use. You shall not have the right to extract or to Use any functionality of this Software, including any DLLs or other compiled units or Object Code fragments other than as part of normal Operation of the Product described in the Documentation and as integral part of Operation and functionality of the Product as a whole.

6.3. Proprietary Notices and Copies. You may not remove any proprietary notices or labels on the Product. You may not copy the Product except as expressly permitted in Section 2 above.

6.4. Compliance with Law. You agree that in Operating the Product and in using any report or information derived as a result of Operating this Product, you will comply with all applicable international, national, state, regional and local laws and regulations, including, without limitation, privacy, trademark, patent, copyright, export control and obscenity law and you shall not use the Product for unethical or illegal business practices or in violation of any obligation to a third party in using, Operating, accessing or running any of the Product and shall not knowingly assist any other person or entity to so violate any obligation to a third party.

7. WARRANTIES AND DISCLAIMERS.

7.1. Limited Warranty. The Licensor warrants that for two (2) months (the ***"Warranty Period"***) from the date the Product has been downloaded by you or was made otherwise available to you by Licensor if the Product was supplied to you on other media, the media on which Product has been provided will be free from defects in materials and workmanship and that the Software will perform substantially in accordance with the Documentation or generally conform to the Product's specifications published by the Licensor. Non-substantial variations of performance from the Documentation do not establish a warranty right. THIS LIMITED WARRANTY DOES NOT APPLY TO UPDATES AS APPLIED TO ANY MODIFIED PRODUCT, WHETHER OR NOT SUCH MODIFICATION IS PERMISSIBLE HEREUNDER, TRIAL AND DEMO OR EVALUATION VERSIONS, UPDATES, PRE-RELEASE, TRYOUT, PRODUCT SAMPLER, OR NOT FOR RESALE (NFR) COPIES OF PRODUCT. This limited warranty is void and your support right terminates if the defect or damage has resulted from accident, abuse, or misapplication or any modification, whether or not such modification is permitted hereunder. Notwithstanding anything to the contrary herein, any and all modifications shall be deemed derivative works within the meaning of copyright law and remain subject to the terms of this Agreement, including without limitation Section 1.1 hereof, *provided however*, that any damage or claims relating to or arising out of any modifications made by Licensee, whether or not permitted hereunder, shall be the sole responsibility of Licensee. To make a warranty claim, you must return the Product to the location where you obtained it along with proof of purchase within such sixty (60) day period of the license fee you paid for the Product. THE LIMITED WARRANTY SET FORTH IN THIS SECTION GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE ADDITIONAL RIGHTS WHICH VARY FROM JURISDICTION TO JURISDICTION.

7.2. Customer Remedies. The Licensor and its suppliers' entire liability and your exclusive remedy for any breach of the foregoing warranty shall be at the Licensor's option: (i) return of the purchase price paid for the License, if any, (ii) replacement of the defective media in which the Product is contained, or (iii) correction of the defects, "bugs" or errors within reasonable period of time. You must return the defective media to the place of purchase at your expense with a copy of your receipt. Any replacement media will be warranted for the remainder of the original Warranty Period.

7.3. NO OTHER WARRANTIES. EXCEPT FOR THE FOREGOING LIMITED WARRANTY, AND FOR ANY WARRANTY, CONDITION, REPRESENTATION OR TERM TO THE EXTENT TO WHICH THE SAME CANNOT OR MAY NOT BE EXCLUDED OR LIMITED BY LAW APPLICABLE TO YOU IN YOUR JURISDICTION, THE PRODUCT IS PROVIDED "AS-IS" WITHOUT ANY WARRANTY WHATSOEVER AND THE LICENSOR

MAKES NO PROMISES, REPRESENTATIONS OR WARRANTIES, WHETHER EXPRESSED OR IMPLIED, WHETHER BY STATUTE, COMMON LAW, CUSTOM, USAGE OR OTHERWISE, REGARDING OR RELATING TO THE PRODUCT OR CONTENT THEREIN OR TO ANY OTHER MATERIAL FURNISHED OR PROVIDED TO YOU PURSUANT TO THIS AGREEMENT OR OTHERWISE. YOU ASSUME ALL RISKS AND RESPONSIBILITIES FOR SELECTION OF THE PRODUCT TO ACHIEVE YOUR INTENDED RESULTS, AND FOR THE INSTALLATION OF, USE OF, AND RESULTS OBTAINED FROM THE PRODUCT. THE LICENSOR MAKES NO WARRANTY THAT THE PRODUCT WILL BE ERROR FREE OR FREE FROM INTERRUPTION OR FAILURE, OR THAT IT IS COMPATIBLE WITH ANY PARTICULAR HARDWARE OR SOFTWARE. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, LICENSOR DISCLAIMS ALL WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT OF THIRD PARTY RIGHTS, INTEGRATION, SATISFACTORY QUALITY OR FITNESS FOR ANY PARTICULAR PURPOSE WITH RESPECT TO THE PRODUCT AND THE ACCOMPANYING WRITTEN MATERIALS OR THE USE THEREOF. SOME JURISDICTIONS DO NOT ALLOW LIMITATIONS ON IMPLIED WARRANTIES, SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU. YOU HEREBY ACKNOWLEDGE THAT THE PRODUCT MAY NOT BE OR BECOME AVAILABLE DUE TO ANY NUMBER OF FACTORS INCLUDING WITHOUT LIMITATION PERIODIC SYSTEM MAINTENANCE, SCHEDULED OR UNSCHEDULED, ACTS OF GOD, TECHNICAL FAILURE OF THE SOFTWARE, TELECOMMUNICATIONS INFRASTRUCTURE, OR DELAY OR DISRUPTION ATTRIBUTABLE TO VIRUSES, DENIAL OF SERVICE ATTACKS, INCREASED OR FLUCTUATING DEMAND, AND ACTIONS AND OMISSIONS OF THIRD PARTIES. THEREFORE, THE LICENSOR EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY REGARDING SYSTEM AND/OR SOFTWARE AVAILABILITY, ACCESSIBILITY, OR PERFORMANCE. THE LICENSOR DISCLAIMS ANY AND ALL LIABILITY FOR THE LOSS OF DATA DURING ANY COMMUNICATIONS AND ANY LIABILITY ARISING FROM OR RELATED TO ANY FAILURE BY THE LICENSOR TO TRANSMIT ACCURATE OR COMPLETE INFORMATION TO YOU.

7.4. LIMITED LIABILITY; NO LIABILITY FOR CONSEQUENTIAL DAMAGES. YOU ASSUME THE ENTIRE COST OF ANY DAMAGE RESULTING FROM YOUR USE OF THE PRODUCT AND THE INFORMATION CONTAINED IN OR COMPILED BY THE PRODUCT, AND THE INTERACTION (OR FAILURE TO INTERACT PROPERLY) WITH ANY OTHER HARDWARE OR SOFTWARE WHETHER PROVIDED BY THE LICENSOR OR A THIRD PARTY. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL THE LICENSOR OR ITS SUPPLIERS OR LICENSORS BE LIABLE FOR ANY DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL OR INDIRECT DAMAGES, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, LOSS OF DATA, LOSS OF GOODWILL, WORK STOPPAGE, HARDWARE OR SOFTWARE DISRUPTION IMPAIRMENT OR FAILURE, REPAIR COSTS, TIME VALUE OR OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OR INABILITY TO USE THE PRODUCT, OR THE INCOMPATIBILITY OF THE PRODUCT WITH ANY HARDWARE SOFTWARE OR USAGE, EVEN IF SUCH PARTIES HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT WILL LICENSOR'S TOTAL LIABILITY TO YOU FOR ALL DAMAGES IN ANY ONE OR MORE CAUSE OF ACTION, WHETHER IN CONTRACT, TORT OR OTHERWISE EXCEED THE AMOUNT PAID BY YOU FOR THE PRODUCT. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO LIABILITY FOR DEATH OR PERSONAL INJURY TO THE EXTENT THAT APPLICABLE LAW PROHIBITS SUCH LIMITATION. FURTHERMORE, BECAUSE SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

8. Indemnification

8.1. Indemnification for Violations. In Operating the Product, you agree to use only those materials for which you have the necessary patent, copyright and other permissions, licenses, and/or clearances. You agree to indemnify, defend and hold harmless the Licensor and its respective officers, directors, employees, agents, successors, and assigns (the ***"Licensor Indemnitees"***) from any and all losses, liabilities, damages and claims, and all related expenses including without limitation reasonable legal fees and disbursements and costs of investigation, litigation, settlement, judgment, interest and penalties and costs related to, arising from, or in connection with any third-party claim related to, arising from, or in connection with the actual or alleged: (i) infringement by you or by Compiled Units (except when such breach is exclusively attributable to Product) of any third-party intellectual property and/or proprietary right, including, but not limited to, patent, trademark, copyright, trade secret, publicity and/or privacy, (ii) personal injury (including death) or property damage due to gross negligence or intentional misconduct of the Licensee, and (iii) breach by the Licensee of any of its representations, warranties, obligations, and/or covenants set forth herein. You shall promptly notify the Licensor in writing after you become aware of any such claims, but failure to give such notice shall not relieve you of indemnity obligations hereunder. You shall have exclusive control over the settlement or defense

of such claims or actions, except that Licensor may appear in the action, at its own expense, through counsel reasonably acceptable to you, only in the event it is mutually determined by the parties that an actual conflict of interest would exist by your representation of the Licensor and you in such action. Licensor shall give you, at your expense, all information and assistance reasonably requested by you to settle or defend such claims or actions. You shall be entitled to retain all monetary proceeds, attorneys' fees, costs and other rewards you receive as a result of defending or settling such claims. In the event you fail to promptly indemnify and defend such claims and/or pay Licensor's expenses, as provided above, Licensor shall have the right to defend itself, and in that case, you shall reimburse the Licensor Indemnitees for all of their attorneys' fees, costs and damages incurred in settling or defending such claims within thirty (30) days of each of Licensor's written requests. Nothing in this Section 3.2 or this Agreement shall be interpreted as to exclude any possible legal recourse against the Licensee.

9. U.S. Government-Restricted Rights.

9.1. Notice to U.S. Government End Users. The Product and accompanying Documentation are deemed to be "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," respectively, as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights, including any use, modification, reproduction, release, performance, display or disclosure of the Product and accompanying Documentation, as are granted to all other end users pursuant to the terms and conditions herein. Unpublished rights are reserved under the copyright laws of the United States.

9.2. Export Restrictions. You acknowledge and agree that the Product may be subject to restrictions and controls imposed by the Export Administration Act and the Export Administration Regulations of the United States (the ***"Acts"***). You agree and certify that neither the Product nor any direct product thereof is being or will be used for any purpose prohibited by the Acts. You may not Operate, download, export, or re-export the Product (a) into, or to a national or resident of, any country to which the United States has embargoed goods, or (b) to anyone on the United States Treasury Department's list of Specially Designated Nationals or the U.S. Commerce Department's Table of Deny Orders. By downloading or using the Product, you are representing and warranting that you are not located in, under the control of, or a national or resident of any such country or on any such list. You acknowledge that it is your sole responsibility to comply with any and all government export and other applicable laws and that the Licensor has no further responsibility for such after the initial license to you. You warrant and represent that neither the U.S. Commerce Department, Bureau of Export Administration nor any other U.S. federal agency has suspended, revoked or denied your export privileges. For more information on the U.S. Export Administration Regulations (EAR), 15 C.F.R. Parts 730-774, and the Bureau of Export Administration (***"BXA"***), please see the BXA homepage (http://www.bxa.doc.gov).

10. Your Information and the Licensor's Privacy Policy

10.1. Privacy Policy. You acknowledge receipt of and agree to the Licensor's privacy statement which is made available to you in connection with installation and is set forth in full at http://www.microolap/about/privacy/. You hereby expressly consent to the Licensor's processing of your personal data (which may be collected by the Licensor or its distributors) according to the Licensor's current privacy policy as of the date of the effectiveness hereof which is incorporated into this Agreement by reference. By entering into this Agreement, you agree that the Licensor may collect and retain information about you, including your name, email address and credit card information. The Licensor may employ other companies and individuals to perform certain functions on its behalf. Examples include fulfilling orders, delivering packages, sending postal mail and e-mail, removing repetitive information from customer lists, analyzing data, providing marketing assistance, processing credit card payments, implementing fraud check policies, and providing customer service. Such companies and individuals may have access to personal information needed to perform their functions, but may not use it for other purposes. The Licensor publishes a privacy policy on the Licensor Site and may amend such policy from time to time in its sole discretion. You should refer to the Licensor's privacy policy prior to agreeing to this Agreement for a more detailed explanation of how your information will be stored and used by the Licensor. If "you" are an organization, you will ensure that each member of your organization (including employees and contractors) about whom personal data may be provided to the Licensor has

given his or her express consent to the Licensor's processing of such personal data. Personal data will be processed by the Licensor or its distributors in the country where it was collected. The relevant laws in such jurisdictions regarding processing of personal data may be less or more stringent than the laws in your jurisdiction.

10.2. Public Announcements. The Licensor may identify you to the public as a customer of the Licensor and describe in a customer case study the services and solutions delivered by the Licensor to you. The Licensor may also issue one or more press releases, containing an announcement of the execution and delivery of this Agreement and/or the implementation of the Product by you. Nothing contained in this Section 10.2 shall be construed as an obligation by you to disclose any of your proprietary or confidential information to any third party. In addition, you may opt-out from this Section 10.2 by writing an opt-out request to the Licensor at sales@microolap.com.

11. Miscellaneous.

11.1. Governing Law; Jurisdiction and Venue. This Agreement shall be governed by and construed and enforced in accordance with the laws of the British Virgin Islands without reference to conflicts of law rules and principles. This Agreement shall not be governed by the United Nations Convention on Contracts for the International Sale of Goods, the application of which is expressly disclaimed and excluded. The courts within the British Virgin Islands shall have exclusive jurisdiction to adjudicate any dispute arising out of this Agreement. You agree that this Agreement and any action, dispute, controversy, or claim that may be instituted based on this Agreement, or arising out of or related to this Agreement or any alleged breach thereof, shall be prosecuted exclusively in the courts of the British Virgin Islands and you, to the extent permitted by applicable law, hereby waive the right to change venue to any other state, county, district or jurisdiction; *provided, however*, that the Licensor as claimant shall be entitled to initiate proceedings in any court of competent jurisdiction.

11.2. Period for Bringing Actions. No action, regardless of form, arising out of the transactions under this Agreement, may be brought by either party hereto more than one (1) year after the cause of action has occurred, or was discovered to have occurred, except that an action for infringement of intellectual property rights may be brought within the maximum applicable statutory period.

11.3. Entire Agreement; Severability; No Waiver. This Agreement is the entire agreement between you and Licensor and supersedes any other prior agreements, proposals, communications or advertising, oral or written, with respect to the Product or to subject matter of this Agreement. You acknowledge that you have read this Agreement, understand it and agree to be bound by its terms. If any provision of this Agreement is found by a court of competent jurisdiction to be invalid, void, or unenforceable for any reason, in whole or in part, such provision will be more narrowly construed so that it becomes legal and enforceable, and the entire Agreement will not fail on account thereof and the balance of the Agreement will continue in full force and effect to the maximum extent permitted by law or equity while preserving, to the fullest extent possible, its original intent. No waiver of any provision or condition herein shall be valid unless in writing and signed by you and an authorized representative of Licensor provided that no waiver of any breach of any provisions of this Agreement will constitute a waiver of any prior, concurrent or subsequent breach. Licensor's failure to insist upon or enforce strict performance of any provision of this Agreement or any right shall not be construed as a waiver of any such provision or right.

11.4. Contact Information. Should you have any questions concerning this Agreement, or if you desire to contact the Licensor for any reason, please contact our Customer Department at http://microolap.com/support/.