DBISAM Version 4 Manual

Table Of Contents

Chapter 1 - Before You Begin **1**
1.1 Changes From Version 3.x 1
1.2 New Features in Version 4.x 10
Chapter 2 - Using DBISAM **17**
2.1 DBISAM Architecture 17
2.2 Data Types and NULL Support 25
2.3 Exception Handling and Errors 31
2.4 Configuring and Starting the Server 35
2.5 Server Administration 44
2.6 Customizing the Engine 50
2.7 Starting Sessions 60
2.8 Calling Server-Side Procedures 64
2.9 Opening Databases 66
2.10 Transactions 68
2.11 Backing Up and Restoring Databases 72
2.12 In-Memory Tables 76
2.13 Creating and Altering Tables 77
2.14 Upgrading Tables 84
2.15 Deleting Tables 86
2.16 Renaming Tables 87
2.17 Adding and Deleting Indexes from a Table 88
2.18 Emptying Tables 91
2.19 Copying Tables 92
2.20 Optimizing Tables 94
2.21 Verifying and Repairing Tables 96
2.22 Opening Tables 99
2.23 Closing Tables 104
2.24 Executing SQL Queries 105
2.25 Live Queries and Canned Queries 112
2.26 Parameterized Queries 114
2.27 Navigating Tables and Query Result Sets 117

2.28 Updating Tables and Query Result Sets 119
2.29 Searching and Sorting Tables and Query Result Sets 128
2.30 Setting Ranges on Tables 134
2.31 Setting Master-Detail Links on Tables 136
2.32 Setting Filters on Tables and Query Result Sets 139
2.33 Loading and Saving Streams with Tables and Query Result Sets 142
2.34 Importing and Exporting Tables and Query Result Sets 144
2.35 Cached Updates 148
Chapter 3 - Advanced Topics 151
3.1 Locking and Concurrency 151
3.2 Buffering and Caching 157
3.3 Change Detection 159
3.4 Index Compression 161
3.5 Filter Optimization 163
3.6 Multi-Threaded Applications 167
3.7 Full Text Indexing 170
3.8 Compression 174
3.9 Encryption 175
3.10 Recompiling the DBISAM Source Code 176
3.11 Replacement Memory Manager 178
Chapter 4 - SQL Reference 179
4.1 Overview 179
4.2 Naming Conventions 180
4.3 Unsupported SQL 190
4.4 Optimizations 192
4.5 Operators 202
4.6 Functions 211
4.7 SELECT Statement 239
4.8 INSERT Statement 253
4.9 UPDATE Statement 255
4.10 DELETE Statement 259
4.11 CREATE TABLE Statement 263
4.12 CREATE INDEX Statement 270
4.13 ALTER TABLE Statement 272
4.14 EMPTY TABLE Statement 275
4.15 OPTIMIZE TABLE Statement 276

4.16 EXPORT TABLE Statement 277
4.17 IMPORT TABLE Statement 279
4.18 VERIFY TABLE Statement 281
4.19 REPAIR TABLE Statement 282
4.20 UPGRADE TABLE Statement 283
4.21 DROP TABLE Statement 284
4.22 RENAME TABLE Statement 285
4.23 DROP INDEX Statement 286
4.24 START TRANSACTION Statement 287
4.25 COMMIT Statement 288
4.26 ROLLBACK Statement 289
Chapter 5 - Component Reference 291
5.1 EDBISAMEngineError Component 291
5.2 TDBISAMBaseDataSet Component 307
5.3 TDBISAMBlobStream Component 309
5.4 TDBISAMDatabase Component 315
5.5 TDBISAMDataSet Component 346
5.6 TDBISAMDataSetUpdateObject Component 388
5.7 TDBISAMDBDataSet Component 390
5.8 TDBISAMEngine Component 399
5.9 TDBISAMFieldDef Component 560
5.10 TDBISAMFieldDefs Component 577
5.11 TDBISAMFunction Component 586
5.12 TDBISAMFunctionParam Component 592
5.13 TDBISAMFunctionParams Component 596
5.14 TDBISAMFunctions Component 602
5.15 TDBISAMIndexDef Component 610
5.16 TDBISAMIndexDefs Component 620
5.17 TDBISAMParam Component 629
5.18 TDBISAMParams Component 663
5.19 TDBISAMQuery Component 672
5.20 TDBISAMRecord Component 723
5.21 TDBISAMSession Component 738
5.22 TDBISAMSQLUpdateObject Component 859
5.23 TDBISAMStringList Component 861
5.24 TDBISAMTable Component 866

5.25 TDBISAMUpdateSQL Component 955

Chapter 6 - Type Reference 967

6.1 TAbortErrorEvent Type 967

6.2 TAbortProgressEvent Type 968

6.3 TCachedUpdateErrorEvent Type 969

6.4 TCompressEvent Type 970

6.5 TCryptoInitEvent Type 971

6.6 TCryptoResetEvent Type 972

6.7 TCustomFunctionEvent Type 973

6.8 TDatabaseRights Type 974

6.9 TDataLostEvent Type 975

6.10 TDecompressEvent Type 976

6.11 TDecryptBlockEvent Type 977

6.12 TEncryptBlockEvent Type 978

6.13 TEndTransactionTriggerEvent Type 979

6.14 TErrorEvent Type 980

6.15 TEventDays Type 981

6.16 TEventMonths Type 982

6.17 TLogEvent Type 983

6.18 TLoginEvent Type 984

6.19 TPasswordEvent Type 985

6.20 TProcedureProgressEvent Type 986

6.21 TProcedureRights Type 987

6.22 TProgressEvent Type 988

6.23 TReconnectEvent Type 989

6.24 TRecordLockTriggerEvent Type 990

6.25 TSendReceiveProgressEvent Type 991

6.26 TServerConnectEvent Type 992

6.27 TServerDisconnectEvent Type 993

6.28 TServerLogCountEvent Type 994

6.29 TServerLogEvent Type 995

6.30 TServerLoginEvent Type 996

6.31 TServerLogoutEvent Type 997

6.32 TServerLogRecordEvent Type 998

6.33 TServerProcedureEvent Type 999

6.34 TServerReconnectEvent Type 1000

6.35 TServerScheduledEvent Type 1001
6.36 TSQLTriggerEvent Type 1002
6.37 TStartTransactionTriggerEvent Type 1003
6.38 TSteppedProgressEvent Type 1004
6.39 TTextIndexFilterEvent Type 1005
6.40 TTextIndexTokenFilterEvent Type 1006
6.41 TTimeoutEvent Type 1007
6.42 TTraceEvent Type 1008
6.43 TTriggerEvent Type 1009
Appendix A - Differences from the BDE 1011
Appendix B - Error Codes and Messages 1023
Appendix C - System Capacities 1041

This page intentionally left blank

Chapter 1
Before You Begin

1.1 Changes From Version 3.x

The following items have been changed in Version 4.x from Version 3.x:

- The physical table format has changed for version 4 and all tables in 3.x and earlier formats will require upgrading to the current format using the TDBISAMTable UpgradeTable method or the new UPGRADE TABLE SQL statement. Please see the Upgrading Tables topic for more information.

  The major changes to the format include:

Change	Description
Table Signatures	Every table is now stamped with an MD5 hash that represents the hash of a "signature" that is specified in the EngineSignature property of the TDBISAMEngine component. In order to access any table, stream, or backup created with a specific engine signature other than the default requires that the engine be using the same signature or else access will be denied. Please see the Customizing the Engine topic for more information.
Locale IDs	The language ID and sort ID values (Word values) for a table in 3.x and lower have been replaced with one single locale ID (Integer value). This causes a change in the TDBISAMTable RestructureTable method, which has been renamed to the AlterTable method to maintain consistency with the ALTER TABLE SQL statement (see below). Also, the LanguageID and SortID properties of the TDBISAMTable component are now one LocaleID property. Finally, the SQL LANGUAGE ID and SORT ID keywords have been replaced with the single LOCALE keyword in SQL statements, and some of the language identifiers (string values) have been modified to reflect the change to a locale instead of a language identifier.
Table Encryption	The default table encryption in prior versions of DBISAM was weak XOR encryption and, although it was fast, it was also easily broken. The table encryption in version 4 is Blowfish encryption that is not easily broken. All table passwords are stored as MD5 hashes encrypted with the same Blowfish encryption. Please see the Encryption topic for more information.
System Fields	There are two new "system" pseudo-fields in every table called "RecordID and "RecordHash". These fields can be indexed, filtered, etc. but do not show up in the field definitions for the TDBISAMTable or TDBISAMQuery components. RecordID is an integer value (4 bytes) representing the fixed "row number" of a given record. RecordHash is an MD5 binary value (16 bytes) that

	represents the hash of a given record. If you upgrade a table that already has a field named the same as either of these fields, your field will be automatically renamed by the UpgradeTable method or the UPGRADE TABLE SQL statement to '_'+OldFieldName. In other words, an underscore will be added to the front of the existing field name.
Auto Primary Index	In version 3.x and earlier you could have a table without a primary index. In version 4, if you do not define a primary index when creating or restructuring a table, DBISAM will automatically add a primary index on the system RecordID field mentioned above.
BLOB Compression	You may now specify compression for BLOB fields when creating or restructuring a table. The compression is specified as a Byte value between 0 and 9, with the default being 0, or none, and 6 being the best selection for size/speed. The default compression is ZLib, but can be replaced by using the TDBISAMEngine events for specifying a different type of compression. Please see the Compression and Customizing the Engine topics for more information.
Maximum Field Size	The maximum size of a string or bytes field is now 512 bytes instead of 250 bytes.
FixedChar Fields	String fields that are of the ftFixedChar type do not automatically right-trim spaces from strings assigned to them as they have in the past. String fields that are of the type ftString still treat strings like VarChars and right-trim the strings assigned to them. For example, assigning the value 'Test ' to the two different field types would result in the following: ftString='Test' ftFixedChar='Test ' This is useful for situations where you want to keep trailing spaces in string fields.
GUID Fields	GUID fields are now supported and are implemented as a 38-byte field containing a GUID in string format.
AutoInc Fields	Auto-increment fields are now always editable and you may have more than one autoinc field per record, with each field incrementing independently. Because these fields are editable, the SuppressAutoIncValues property has been removed from both the TDBISAMTable and TDBISAMQuery component and the NOAUTOINC clause has been removed from the SQL statements. The way autoinc fields work now is that they will auto-increment if a value is not specified for the field before the Post operation (field is NULL), and will leave any existing value alone if one is already specified before the Post operation.

	Note If you do not want an end user to modify any autoinc fields directly then it is extremely important that you mark any autoinc fields as read-only by setting the TField ReadOnly property to True before the user is allowed to access these fields.
Descending Index Fields	You may now specify which fields are ascending or descending in an index independently of one another. This change also modifies the AddIndex method of the TDBISAMTable component slightly as well as the TDBISAMIndexDef objects used in creating and altering the structure of tables. With SQL you can simply place an appropriate ASC or DESC keyword after each field specified for an index definition in a CREATE TABLE or CREATE INDEX statement.
Index Page Size	You may now specify the index page size when creating or altering the structure of tables. This changes the TDBISAMTable AlterTable method slightly as well as the CREATE TABLE SQL statement syntax. Also, there is a new IndexPageSize property for the TDBISAMTable component. The minimum page size is 1024 bytes and the maximum page size is 16 kilobytes. **Note** The index page size affects the maximum key size that can be specified for an index, so if you try to index very large string fields you may get an error indicating that the index key size is invalid. Also, regardless of page size the maximum key size for any index is 4096 bytes. Finally, the maximum number of fields that can be included in a given index has been expanded from 16 to 128 fields. However, the number of indexes per table is still only 30 indexes and has not changed.

- The TDBISAMTable RestructureTable method is now called the AlterTable method to be more in line with the name of the ALTER TABLE SQL statement. Also, the TDBISAMTable OnRestructureProgress event is now called the OnAlterProgress event.
- The TDBISAMTable OnDataLost event will now fire when adding unique secondary or primary indexes that cause key violations. Also, the ContinueRestructure parameter to this event is now called the Continue parameter in order to be more in line with its new dual-purposes.
- The TDBISAMQuery OnQueryProgress event is now of the type TAbortProgressEvent to reflect the fact that it will be used for more than just the OnQueryProgress event in the future.
- The addition and subtraction of dates, times, and timestamps in filter and SQL expressions have changed slightly. Please see the SQL Reference Operators topic for more information.

- There are also new filter and SQL functions for converting milliseconds into the appropriate number of years, days, hours, etc. Please see the New Features in Version 4.x and the SQL Reference Functions topics for more information.

- The index compression/de-compression code has been vastly improved so as to be much more efficient, especially when there are a large number of duplicate keys in the index and the compression is set to duplicate-byte or full compression.

- The DBISAM table stream format has changed completely. It is now more similar to a binary import/export format and can now include just a subset of fields from the original table and does not include index information that previously caused many problems with loading streams saved from query result sets into tables, etc.

> **Note**
> Like tables themselves, streams are signed with the current engine signature to ensure that only the current engine signature, or the default engine signature, can access the stream. Also, even though the table that a stream is created from is encrypted, the resultant stream will never be encrypted and you must make sure to take extra caution if you do not want to expose data improperly. Please see the Loading and Saving Streams with Tables and Query Result Sets topic for more information.

- The table locking in DBISAM has changed completely in order to streamline transaction locking, prevent deadlocks during transactions, and improve the performance of the table and transaction locking. Previously table locking was done at the individual table level, so if you started a transaction on a database with 50 physical tables opened for that database, DBISAM would have to place a transaction lock on all 50 open tables before starting the transaction. It would also have to subsequently write lock them during a commit and then unlock everything for each table after the transaction was committed or rolled back. Now all table locking is centralized in one hidden file called "dbisam.lck" (by default) and located in the physical database directory. In case anyone mistakes this for a Paradox-style lock file, it is definitely not anything close. The lock file in DBISAM version 4 is just an empty "container" used to perform byte offset locking at the operating system level and the existence of the file is strictly optional - it will automatically be created by DBISAM as needed. Likewise, if the file is left there (which it will be since DBISAM prefers not to have to constantly recreate it when needed) it will not cause any harm, unlike with a Paradox lock file. With this new type of locking, DBISAM only needs to place one lock call to the OS when a transaction is started (instead of the previous scenario of 50 calls), one write lock call during a commit, and one unlock call during a commit or rollback. It also completely eliminates deadlocks during transaction locking since this architecture makes it impossible to get a deadlock. Please see the Locking and Concurrency and Transactions topics for more information.

> **Note**
> The default lock file name "dbisam.lck" can be modified to any file name desired by modifying the TDBISAMEngine LockFileName property.

- A few TDBISAMSession properties have been modified slightly to reflect some changes in the remote access. The RemoteType property has been removed and been replaced with the RemoteEncryption, RemoteEncryptionPassword, and RemoteCompression properties. The RemoteEncryption property specifies that any comms requests or responses should be encrypted using the strong crypto in the engine, and the RemoteEncryptionPassword specifies the password to use for the encryption. This password must match the password used by the server engine to encrypt/decrypt comms on its end. Also, in version 4 *all* administrative access requires the use of RemoteEncryption=True. You cannot log into the administrative port on a server without encryption turned on and the password set to the proper password for the server that you are accessing. In addition to this, all login information is automatically encrypted using the RemoteEncryptionPassword, so regardless of whether RemoteEncryption is turned on or not, the password must still match that of the server or you won't be able to log in using a non-encrypted connection either. The RemoteCompression property allows you to dynamically change the compression for the comms at any time before, during, or after logging into a database server. Each request and response is tagged with a specific compression level, thus allowing unlimited flexibility in determining how much/little compression to use. The property is specified as a Byte value between 0 and 9, with the default being 0, or none, and 6 being the best selection for size/speed. Because of these property changes, the TDBISAMSession GetRemoteSessionInfo method has been modified to reflect whether the session is encrypted or not instead of the type of session (rtInternet or rtLAN previously).

- The TDBISAMSession method GetRemoteLog for retrieving the server log from the server has been removed and replaced with two different methods, one for retrieving the total number of log entries called GetRemoteLogCount, and one for retrieving a specific log entry from the server based upon its ordinal position in the log called GetRemoteLogRecord. This change is due to the abstraction of the log storage in the TDBISAMEngine component when running as a server (EngineType=etServer). Previously the log storage was a "black box" text file that was maintained by the server. Now the log storage is abstract and is handled via the OnServerLogEvent event in the TDBISAMEngine component. A TLogRecord record is passed to an event handler for this event and the event handler is free to store this data in whatever way it deems appropriate. Likewise, the OnServerLogCount event is triggered in the TDBISAMEngine component when the client session calls the TDBISAMSession GetRemoteLogCount method and the OnServerLogRecord event is called when the TDBISAMSession GetRemoteLogRecord method is called.

  Note
  By default, the server application that comes with DBISAM uses event handlers for these events to simply write out these log records as binary records in a log file.

  Please see the Customizing the Engine topic for more information.

- The following types have been changed or removed:

Type	New Type
TDBISAMPasswordEvent	TPasswordEvent
TDBISAMDatabaseRight	TDatabaseRight **Note** The TDatabaseRight type has also been expanded to include new rights for backup (drBackup) and restore (drRestore) of a database, as well as rights for performing maintenance (drMaintain) on a database like repairing and optimizing tables and renaming objects in a database (drRename).
TDBISAMDatabaseRights	TDatabaseRights

- The following constants have been changed or removed:

Constant	New Constant(s)
DBISAM_LOCKTIMEOUT	DBISAM_READLOCK DBISAM_READUNLOCK DBISAM_WRITELOCK DBISAM_WRITEUNLOCK DBISAM_TRANSLOCK DBISAM_TRANSUNLOCK This was done to give the developer more control over which condition he/she was responding to, especially when it comes to transaction lock timeouts.

- The RestructureFieldDefs and RestructureIndexDefs have been removed and replaced with common TDBISAMFieldDefs and TDBISAMIndexDefs objects. These new objects allow the TDBISAMTable CreateTable method to be changed so that it is identical to the AlterTable method (used to be called RestructureTable), thus eliminating the need for the old way of creating a table and then immediately altering its structure in order to add DBISAM-specific features to the table. These objects are assignment-compatible with their TDataSet cousins TFieldDefs and TIndexDefs.

  Note
  There is one important change in the TDBISAMFieldDefs Add method that is different from the standard TFieldDefs Add method. The TDBISAMFieldDefs Add method is overloaded to allow for the direct specification of the FieldNo of the TDBISAMFieldDef being added. This is to allow for moving fields around without losing any data with the AlterTable method. Also, the TDBISAMFieldDefs object has an additional Insert method that allows for the insertion of a TDBISAMFieldDef object in a specific position in the TDBISAMFieldDefs. Please see the Creating and Altering Tables topic for more information.

- The TDBISAMTable and TDBISAMQuery BlockReadSize property functionality has been modified so that it behaves like the TDBISAMTable and TDBISAMQuery RemoteReadSize property, which does not have the limitations that the BlockReadSize property used to have and can also very easily optimize C/S access so that records are retrieved from the server in batches.

- The TDBISAMTable RecordIsLocked and TableIsLocked methods no longer attempt to make locking calls in order to determine whether a record or table is locked, and only reflect whether the current table cursor has a given record or table locked. If you want to edit a record you should just edit the record and respond accordingly to any locking exceptions that occur if a table or record is already locked.

- The TDBISAMTable and TDBISAMQuery Locate method implementation has internally been moved into the engine itself, which should result in some faster performance for Locate calls, especially when accessing a database server. Also, the Locate method can now take advantage of indexes in live query result sets (as well as canned result sets) when optimizing its searching. These changes to Locate do not cause any code changes in your application.

- All DBISAM error strings are now marked with the resourcestring directive and are located in a new unit (Delphi) or header file (C++) called dbisamst.

- The TDBISAMQuery Params property is no longer the standard TParams object, but rather is now a custom TDBISAMParams object. This also holds true for the individual TParam objects contained within the Params property, as they are now TDBISAMParam object. This was done to fix a bug in the parsing of parameters in SQL statements in the TParams object, as well as to enable the use of a common set of objects for both queries, custom SQL and filter functions, and server-side procedure calls. Also, with this change we have added the TDBISAMParam AsLargeInt property to allow you to retrieve and assign 64-bit integer parameters.

- The TDBISAMQuery component now processes SQL scripts client-side so as to allow for the use of parameters with scripts. A new OnGetParams event is fired whenever a new SQL statement is prepared. This allows one to execute an SQL script and populate the parameters in a step-by-step fashion. However, it does come at a price when executing large SQL scripts using a remote session. Previously with 3.x the entire script was executed on the database server, but with version 4 each individual SQL statement is parsed and sent to the server independently, so this can result in much more network traffic. The work-around is to send any very large SQL scripts to the server to be executed in the context of a server-side procedure, which will keep the processing of the script entirely on the server but still allow for parameters in the script.

- SQL statements and filter expressions now require all constants to be enclosed in single quotes as opposed to double-quotes. Identifiers such as table names and column names can still be (and must be) enclosed in double quotes or brackets. This allows DBISAM's parser to distinguish properly between identifiers and constants, which previously would confuse the parser, especially with expressions like this:

```
MyColumName="MyColumnName"
```

  where the parser didn't know whether to treat "MyColumnName" as a constant or a column value.

- The use of the asterisk (*) as a wildcard along with the equality (=) operator in SQL statements is no longer supported. Instead, you must use the LIKE operator and the percent (%) wildcard character like this:

```
MyColumName LIKE 'Test%'
```

- The SQL aggregate and distinct processing, as well as the result set ordering, has been improved so as to reduce the amount of I/O used to perform these functions. The results should be fairly improved over 3.x, especially with large source tables. In addition, the MIN and MAX aggregate functions can now take advantage of indexes when SQL statements like the following are used:

```
SELECT max(MyField) FROM MyTable
```

  where MyTable has an index on MyField. You can also now use the MIN and MAX aggregate functions with string fields. Finally, the SQL SELECT statement's TOP clause can now take advantage of indexes to optimize its performance quite a bit over 3.x.

- The MEMORY keyword has been removed from SQL statements and should be replaced with a database specification of "Memory\". For example, in 3.x you would specify the following SQL SELECT statement to retrieve data from an in-memory table:

```
SELECT * FROM MEMORY biolife
```

  In version 4 you should use:

```
SELECT * FROM "\Memory\biolife"
```

- The WITH LOCKS clause has been removed from the SELECT SQL statement. To ensure that data does not change during the course of a SELECT statement you should wrap the statement in a transaction.
- The SQL and filter LIKE operator now accepts an ESCAPE clause to specify an escape character:

```
SELECT * FROM MyTable WHERE MyColumn LIKE '100\%%' ESCAPE '\'
```

In the above example the backslash serves as the escape character indicating that the character after it, the percent sign (%), should be interpreted literally and not as a wildcard like it normally is. The above SQL statement will find all records where MyColumn begins with '100%'.

1.2 New Features in Version 4.x

The following items are new features in version 4.x:

- There is a new TDBISAMEngine component that encapsulates the DBISAM engine inside of a visual component. In the component hierarchy, the TDBISAMEngine component sits at the top above the TDBISAMSession component(s). A default Engine function is available in the dbisamtb unit (Delphi) or dbisamtb header file (C++) that points to a global instance of the TDBISAMEngine component. You can also drop a TDBISAMEngine component on a form or data-module to visually change its properties. However, only one instance of the TDBISAMEngine component can exist in a given application, and both the Engine function and any TDBISAMEngine component on a form or data module point to the same instance of the component (singleton model). Some of the functionality found in the TDBISAMEngine component includes:

Functionality	Description
Engine Type	The EngineType property can be set to either etClient or etServer in order to have the engine behave as a local client engine or a server engine. If acting as a server engine, many additional properties are provided for configuring the server: ServerName ServerDescription ServerMainAddress ServerMainPort ServerMainThreadCacheSize ServerAdminAddress ServerAdminPort ServerAdminThreadCacheSize ServerEncryptedOnly ServerEncryptionPassword ServerConfigFileName ServerConfigPassword There are also many events provided for the server engine: OnServerStart OnServerStop OnServerLogEvent OnServerLogCount OnServerLogRecord OnServerConnect OnServerReconnect OnServerLogin OnServerLogout OnServerDisconnect OnServerScheduledEvent OnServerProcedure Please see the Configuring and Starting the Server topic for more information.
Full Text Indexing	There are specific events for implementing full text index

	filtering (either on a buffer basis or on a per-token basis): OnTextIndexFilter OnTextIndexTokenFilter Also, there are two new methods for parsing strings into word lists and retrieving the default text indexing parameters: BuildWordList GetDefaultTextIndexParams **Note** The BuildWordList function used to be available in the dbisamlb unit (Delphi) or dbisamlb header file (C++) and it is still is, although different from the one available as a method of the TDBISAMEngine component. You should use the method of the TDBISAMEngine component instead of the function in the dbisamlb unit in version 4. Please see the Full Text Indexing topic for more information.
Custom Encryption	There are specific events for customizing the encryption in DBISAM (8-byte block ciphers only): OnCryptoInit OnEncryptBlock OnDecryptBlock OnCryptoReset Please see the Encryption topic for more information.
Custom Compression	There are specific events for customizing the compression in DBISAM: OnCompress OnDecompress Please see the Compression topic for more information.
Signatures	There is an EngineSignature property in the TDBISAMEngine component that is used to create an MD5 hash that is assigned to every table, table stream, backup, comms request and response, etc. This allows one to "assign" tables, etc. to a specific application and prevent any other application from accessing the tables, server, etc. without the proper engine signature. Please see the Customizing the Engine for more information.
ANSI Conversions	All of the ANSI string conversion functions that used to be in the dbisamlb unit are now public methods of the TDBISAMEngine component:

	DateToAnsiStr TimeToAnsiStr DateTimeToAnsiStr AnsiStrToDate AnsiStrToTime AnsiStrToDateTime BooleanToAnsiStr AnsiStrToBoolean FloatToAnsiStr AnsiStrToFloat CurrToAnsiStr AnsiStrToCurr
Locale Functionality	There are new methods for working with the available locales in DBISAM: IsValidLocale IsValidLocaleConstant ConvertLocaleConstantToID ConvertIDToLocaleConstant GetLocaleNames
Memory Usage	The amount of memory used for buffering tables can now be controlled via the following properties: MaxTableDataBufferSize MaxTableDataBufferCount MaxTableIndexBufferSize MaxTableIndexBufferCount MaxTableBlobBufferSize MaxTableBlobBufferCount **Note** These properties used to be in the TDBISAMSession component in 3.x and earlier and were only applicable to the session for which they were configured. The TDBISAMEngine properties above are used for the all sessions in the application.
File Extensions	The file extensions to use for physical table files, table backup files, and table upgrade backup files can be specified via the following properties: TableDataExtension TableIndexExtension TableBlobExtension TableDataBackupExtension TableIndexBackupExtension TableBlobBackupExtension TableDataUpgradeExtension TableIndexUpgradeExtension TableBlobUpgradeExtension
Locking	The lock wait times and retry counts for table read, write, and transaction locks can now be modified via the

	following properties: TableReadLockTimeout TableWriteLockTimeout TableTransLockTimeout
Triggers	You can now define trigger event handlers that allow for processing both before and after the execution of an insert, update, or delete operation: BeforeInsertTrigger AfterInsertTrigger BeforeUpdateTrigger AfterUpdateTrigger BeforeDeleteTrigger AfterDeleteTrigger Please see the Customizing the Engine topic for more information.
Custom Functions	You can now add custom functions for use with filters and SQL statements. They can be used anywhere that a normal, non-aggregate function would be used. All arguments to the functions are required and there is no facility currently for optional arguments. The Functions property of the TDBISAMEngine component allows you to specify the functions and their arguments, and the OnCustomFunction event of the TDBISAMEngine component allows you to implement the functions. Please see the Customizing the Engine topic for more information.

- You can now use restricted transactions on a given database where only certain tables that you specify are involved in the transaction. Please see the Transactions topic for more information.

- There is a new TDBISAMEngine FilterRecordCounts property that controls how record counts are returned for filtered datasets and live query result sets. The default value of this property is True, which indicates that record counts under these circumstances will be returned in the same fashion as they were in 3.x and earlier. If the FilterRecordCounts property is set to False, the RecordCount property of the TDBISAMTable and TDBISAMQuery components will always show the total record count of the entire dataset or active range (if a range is set) only and will not take any active filters (or WHERE clauses with live query result sets) into account. To get the record count including any active filters, a FilterRecordCount property has been added to the TDBISAMTable and TDBISAMQuery components that always shows the accurate record count, regardless of the current setting of the TDBISAMEngine FilterRecordCounts propety.

  Setting the TDBISAMEngine FilterRecordCounts property to False may be desirable for some applications since it allows for more accurate positioning of the scroll bar in a TDBGrid or similar multi-row, data-aware components. Please see the Customizing the Engine and Setting Filters on Tables topics for more information.

- The TDBISAMSession component now has new remote administrative methods for adding/updating/deleting server-side procedures and events:

  GetRemoteProcedureNames
  GetRemoteProcedure
  AddRemoteProcedure
  ModifyRemoteProcedure
  DeleteRemoteProcedure
  GetRemoteProcedureUserNames
  GetRemoteProcedureUser
  AddRemoteProcedureUser
  ModifyRemoteProcedureUser
  DeleteRemoteProcedureUser
  GetRemoteEventNames
  GetRemoteEvent
  AddRemoteEvent
  ModifyRemoteEvent
  DeleteRemoteEvent

  Please see the Server Administration topic for more information.

- The TDBISAMSession component now has the ability to ping a database server using the RemotePing and RemotePingInterval properties. These properties eliminate the need for user-constructed pinging operations using timers and are safe to use for the purpose of shortening dead session expiration times that are configured on a database server and eliminating dangling pessimistic locks when client workstations go down while connected.

- The TDBISAMSession component now has the capability to call a server-side procedure on a database server using the CallRemoteProcedure method, the RemoteParams property, and the RemoteParamByName method. Please see the Calling Server-Side Procedures topic for more information.

- The TDBISAMDatabase component has new backup and restore facilities available in the following methods and events:

  Backup
  BackupInfo
  Restore

  OnBackupProgress
  OnBackupLog
  OnRestoreProgress
  OnRestoreLog

  Please see the Backing Up and Restoring Databases topic for more information.

- There is a new TableSize property for the TDBISAMTable component that reflects the total size (in bytes) of the physical table on disk (or in-memory if an in-memory table).

- The SQL SELECT statement now includes support for the EXCEPT [ALL] and INTERSECT [ALL] set operations, in addition to the UNION [ALL] operation.

- There are several new SQL statements available:

  EMPTY TABLE
  OPTIMIZE TABLE
  EXPORT TABLE
  IMPORT TABLE
  VERIFY TABLE
  REPAIR TABLE
  UPGRADE TABLE
  RENAME TABLE

- There are several new filter and SQL functions:

  STDDEV (aggregate, SQL-only)
  CURRENT_GUID
  YEARSFROMMSECS
  DAYSFROMMSECS
  HOURSFROMMSECS
  MINSFROMMSECS
  SECSFROMMSECS
  MSECSFROMMSECS
  LTRIM
  RTRIM
  REPEAT
  CONCAT
  MOD
  ACOS
  ASIN
  ATAN
  ATAN2
  CEILING or CEIL
  COS
  COT
  DEGREES
  EXP
  FLOOR
  LOG
  LOG10
  PI
  POWER
  RADIANS
  RAND
  SIGN
  SIN
  SQRT
  TAN
  TRUNCATE or TRUNC

  Please see the SQL Reference Functions topic for more information.

- The SQL engine can now use the numeric 1 (or anything not 0) and 0 to represent TRUE and FALSE, respectively. This is helpful for compatibility with generic front ends, such as those used with the ODBC driver.

- There is a new TDBISAMQuery OnQueryError event that can be used to trap SQL errors and decide whether to abort an executing SQL statement or not. If an OnQueryError event handler is not assigned, then any SQL errors will immediately surface as an EDBISAMEngineError exception in the TDBISAMQuery component.

- The TDBISAMQuery component now surfaces the OnAlterProgress, OnDataLost, OnIndexProgress, OnOptimizeProgress, OnRepairLog, OnRepairProgress, OnUpgradeLog, OnUpgradeProgress, OnVerifyLog, and OnVerifyProgress events just like the TDBISAMTable component. The only difference is these events are triggered when the corresponding SQL statement is executed instead of being triggered by a method call, including situations where an SQL statement is executed within a script.

- There are new OnLoadFromStreamProgress and OnSaveToStreamProgress events in the TDBISAMTable and TDBISAMQuery components for tracking the loading/saving progress of streams.

Chapter 2
Using DBISAM

2.1 DBISAM Architecture

Introduction

DBISAM is a database engine that can be compiled directly into your Delphi or C++ application, be it a program or library, or it can be distributed as a runtime package (equivalent to a library) as part of your application. DBISAM was written in Delphi's Object Pascal and can be used with the VCL (Windows only).

General Architecture

DBISAM itself is a lightweight engine encapsulated within the TDBISAMEngine component. When the TDBISAMEngine EngineType property is set to etClient, the TDBISAMEngine component is acting as a local client engine, and when the EngineType property is set to etServer, the TDBISAMEngine component is acting as a database server.

Sessions

DBISAM is session-based, where a session is equivalent to a virtual user and is encapsulated within the TDBISAMSession component. There can be many sessions active in a given application, such as is the case with a multi-threaded application. In multi-threaded applications DBISAM requires a separate session for each thread performing database access. Please see the Multi-Threaded Applications topic for more information.

A DBISAM session can be either local or remote:

Session Type	Description
Local	A local session gains direct access to database tables via the operating system API to a given storage medium, which can literally be any such medium that is accessible from the operating system in use. This means that a local session on the Windows operating system could access database tables on a Linux file server. DBISAM automatically provides for the sharing of database tables using a local session. For example, an application can use local sessions on a small peer-to-peer network to provide a low-cost, multi-user solution without the added expense of using the client-server version of DBISAM. A local session has all of the capabilities of a remote session except for user and database security, which are only available from a database server. Also, with a local session a directory is synonymous with a database, whereas with a remote session databases are defined as part of the server configuration and the DBISAM client does not know the actual location of a given database.
Remote	A remote session uses sockets to communicate to a database server over a network (or on the same physical machine) using the TCP/IP protocol. DBISAM allows a remote session to

	be entirely encrypted using strong crypto. Compression is also available for remote sessions and can be changed whenever it is deemed necessary in order to improve the data transfer speed. This is especially important with low-bandwidth connections like a dial-up Internet connection. A remote session connects to a given database server via an IP address or host name and one of two different ports, depending upon whether the connection is a regular connection or an administrative connection. Before a remote session can perform any operation on a database server it must be logged in with a proper user ID and password. If a remote session is connecting to the administration port on a database server, the user ID specified during the login must be that of an administrator or the login will be rejected. Also, an administrative connection must be encrypted or the database server will reject the connection.

Note
A developer can mix as many local and remote sessions in one application as needed, thus enabling a single application to access data from a local hard drive, a shared file server, or a database server. Also, local and remote sessions are completely identical from a programming perspective, offering both navigational and SQL access methods. The only changes needed to switch from local access to remote access for a session component is the modification of the TDBISAMSession SessionType property.

Database Server

The database server listens for regular data connections on one port and administrative connections on a second port. All administrative connections must be encrypted or they will be rejected by the database server. When the TDBISAMEngine Active property is set to True, the database server will start listening on the IP addresses and ports indicated by the following properties:

ServerMainAddress
ServerMainPort
ServerAdminAddress
ServerAdminPort

If the either ServerMainAddress or ServerAdminAddress property is blank (the default), the database server will listen on all IP addresses available for the type of connection (either regular or administrative). The default ports are 12005 for the ServerMainPort property and 12006 for the ServerAdminPort property. Once the server is started, you cannot change any of these properties, as well as several other properties. Please see the Configuring and Starting the Server topic for more information.

The database server is a multi-threaded server that uses one thread per client connection, which corresponds to a client TDBISAMSession component set to run as a remote session via the SessionType property. DBISAM will cache threads and keep a pool of unused threads available in order to improve connect/disconnect times. The following properties control the default thread cache size uses by the database server:

ServerMainThreadCacheSize
ServerAdminThreadCacheSize

The default for the ServerMainThreadCacheSize property is 10 threads and the default for the

ServerAdminThreadCacheSize property is 1. Both of these properties must be set before the engine is started and cannot be changed when the engine is started.

"Dead" sessions in the database server are sessions that have been inactive for a connection timeout period (configurable) due to lack of client session requests or due to a physical network interruption in service. Such sessions retain their complete state from the time that the disconnect occurred. The sessions remain in this state until:

- The client session attempts another data request or pings the server, in which case the connection will automatically be re-established transparently between the client session and the database server.
- The database server's dead session expiration time period (configurable) is reached and the database server automatically removes the session.
- The number of dead sessions on the database server reaches the maximum threshhold (configurable), thus causing the database server to remove dead sessions in order to bring the number back under the threshhold, oldest dead session first.

Note
The age of a dead session is determined by the last time that the session was connected to the server.

Please see the Server Administration topic for more information on configuring these settings on the server.

Note
You can configure the remote sessions on the client to ping the database server at regular intervals via the TDBISAMSession RemotePing and RemotePingInterval properties. Configuring remote sessions to ping the database server in a smaller time period than the connection timeout configuration on the database server allows you to specify a smaller dead session expiration timeout and prevent sessions with active locks from being left around for too long. With pinging turned on, the only reason a session would be disconnected by the server is if the client workstation or the physical network connection has failed.

You may have a database server (or several) accessing a given database at the same time as other local applications such as CGI or ISAPI web server applications. This allows you to put critical server-side processing on the server where it belongs without incurring a lot of unnecessary overhead that would be imposed by the transport protocol of the database server. This can improve the performance of server-based local applications significantly, especially when they reside on the same machine as the database server and the databases being accessed are local to the server machine.

The database server allows you to configure all users, databases, server-side procedures, and scheduled events via a remote administrative connection or directly via the TDBISAMEngine component. User security at the database and server-side procedure level allows the configuration of read, execute, insert, update, delete, create, alter, drop, rename, maintain, backup, and restore privileges for a specific user or users. Additionally, you may allow or block specific IP addresses or ranges of IP addresses (using wildcards) for access to a given database server. A maximum number of connections may be set to prevent too many inbound connections to a given server. Because the database server does not actively establish any communication with a client session and all communication is controlled by the client session, you do not have issues with firewalls as long as the firewall allows for inbound access to the main port and/or administration port on the server. Please see the Server Administration topic for more information.

All connections, errors, and other operational messages are logged and can be retrieved at a later time by an administrator for examination.

Databases and Directories

DBISAM uses the physical directories in the operating system's file system to represent databases. This is true for both local sessions and remote sessions, however with remote sessions these directories are abstracted through logical database names in the server configuration. This allows applications written to use remote sessions connecting to a database server to be portable between different servers with different directory layouts. DBISAM creates a single hidden file called "dbisam.lck" (by default) in a database directory that is used for locking. It is created as needed and may be deleted if not in use by DBISAM. However, if DBISAM cannot write to this file it will treat the database as read-only. Please see the Locking and Concurrency topic for more information.

> **Note**
> The default lock file name "dbisam.lck" can be modified to any file name desired by modifying the TDBISAMEngine LockFileName property.

Physical Table Layout

DBISAM tables are divided into up to 3 physical files, one for data records, one for indexes, and one for BLOB data (if there are BLOB fields present in the table):

File Type	Description
Data File	Used to store a fixed-length header for table-wide definitions such as the table description, field counts, autoinc values, etc., the fixed-length field definitions for the table, and the fixed-length data records themselves. The use of a fixed-length header, field definitions, and data records allows for easier verification and/or repair of tables in the case of physical table corruption. Please see the Verifying and Repairing Tables topic for more information. All data records contain a small record header and the field data. BLOB fields contains a link to the BLOB file where the actual variable-length BLOB data is stored in a blocked format.
Index File	Used to store a fixed-length header for index statistics, index counts, etc., the fixed-length index definitions, and the fixed-length index pages themselves. The index page size is variable and can be set between 1024 bytes and 16 kilobytes on a per-table basis. All index pages for all primary, secondary, and full text indexes are stored in this file.
BLOB File	Used to store a fixed-length header for BLOB statistics, etc. and the fixed-length BLOB blocks themselves. The BLOB block size is variable and can be set between 64 bytes and 64 kilobytes on a per-table basis. All BLOB blocks for all BLOB fields are stored in this file.

The file extensions used for these physical files can be changed. Please see the Customizing the Engine topic for more information. The default file extensions are as follows:

File Type	File Extension
Data File	.dat
Index File	.idx
BLOB File	.blb

In addition, during certain operations such as altering a table's structure, backup files will be created for the physical table files. The default backup file extensions are as follows:

File Type	Backup File Extension
Data File	.dbk
Index File	.ibk
BLOB File	.bbk

Finally, during the process of upgrading a table from a previous version's format to the latest format, backup files will be created for the physical table files. The default backup file extensions for upgraded tables are as follows:

File Type	Upgrade Backup File Extension
Data File	.dup
Index File	.iup
BLOB File	.bup

Please see the Upgrading Tables topic for more information.

Component Architecture

DBISAM includes the following components:

Component	Description
TDBISAMEngine	The TDBISAMEngine component encapsulates the DBISAM engine itself. A TDBISAMEngine component is created automatically when the application is started and can be referenced via the global Engine function in the dbisamtb unit (Delphi) and dbisamtb header file (C++). You can also drop a TDBISAMEngine component on a form or data-module to visually change its properties. However, only one instance of the TDBISAMEngine component can exist in a given application, and both the global Engine function and any TDBISAMEngine component on a form or data module point to the same instance of the component (singleton model). The TDBISAMEngine component can be configured so that it acts like a local or client engine (etClient) or a database server via the EngineType property. The engine can be started by setting the Active property to True.
TDBISAMSession	The TDBISAMSession component encapsulates a session in

	DBISAM. A default TDBISAMSession component is created automatically when the application is started and can be referenced via the global Session function in the dbisamtb unit (Delphi) and dbisamtb header file (C++). The TDBISAMSession component can be configured so that it acts like a local (stLocal) or a remote session (stRemote) via the SessionType property. A local session is single-tier in nature, meaning that all TDBISAMDatabase components connected to the session reference directories in a local or network file system via the Directory property and all TDBISAMTable or TDBISAMQuery components access the physical tables directly from these directories using operating system API calls. A remote session is two-tier in nature, meaning that all access is done through the remote session to a database server using the DBISAM messaging protocol over a TCP/IP connection. The database server is specified through the following properties: RemoteHost or RemoteAddress RemotePort or RemoteService In a remote session, all TDBISAMDatabase components reference databases that are defined on the database server via the RemoteDatabase property and all TDBISAMTable or TDBISAMQuery components access the physical tables through the DBISAM messaging protocol rather than directly accessing them. **Note** You cannot activate remote sessions in an application whose TDBISAMEngine component is configured as a database server via the EngineType property. A session can be started by setting the Active property to True or by calling the Open method. The TDBISAMSession component contains a SessionName property that is used to give a session a name within the application. All sessions must have a name before they can be started. The default TDBISAMSession component is called "Default". The TDBISAMDatabase, TDBISAMTable, and TDBISAMQuery components also have a SessionName property and these properties are used to specify which session these components belong to. Setting their SessionName property to "Default" or blank ("") indicates that they will use the default TDBISAMSession component. Please see the Starting Sessions topic for more information.
TDBISAMDatabase	The TDBISAMDatabase component encapsulates a database in DBISAM. It is used as a container for a set of tables in a physical directory for local sessions or as a container for a set of tables in a database on a database server for remote sessions. Please see the Server Administration topic for more information on defining databases on a database server. A database can be opened by setting the Connected property to

	True or by calling the Open method. A TDBISAMDatabase component contains a DatabaseName property that is used to give a database a name within the application. All databases must have a name before they can be opened. The TDBISAMTable and TDBISAMQuery components also have a DatabaseName property and these properties are used to specify which database these components belong to. Please see the Opening Tables topic for more information. The TDBISAMDatabase Directory property indicates the physical location of the tables used by the TDBISAMTable and TDBISAMQuery components. If a TDBISAMDatabase component is being used with a local session (specified via the SessionName property), then its Directory property should be set to a valid physical path for the operating system in use. The TDBISAMDatabase RemoteDatabase property indicates the name of a database defined on a database server. If a TDBISAMDatabase component is connected to a remote session (specified via the SessionName property), then its RemoteDatabase property should be set to a valid database for the database server that the session is connected to. The TDBISAMDatabase component is used for transaction processing via the StartTransaction, Commit, and Rollback methods. Please see the Transactions topic for more information. You can backup and restore databases via the Backup, BackupInfo, Restore methods. Please see the Backing Up and Restoring Databases topic for more information.
TDBISAMTable	The TDBISAMTable component encapsulates a table cursor in DBISAM. It is used to search and update data within the physical table specified by the TableName property, as well as create the table or alter its structure. A table cursor can be opened by setting the Active property to True or by calling the Open method. The DatabaseName property specifies the database where the table resides. Please see the Opening Tables topic for more information. The TDBISAMTable component descends from the TDBISAMDBDataSet component, which descends from the TDBISAMDataSet component, which descends from the common TDataSet component that is the basis for all data access in Delphi and C++. None of these lower-level components should be used directly and are only for internal structuring purposes in the class hierarchy. You can have multiple TDBISAMTable components using the same physical table. Each TDBISAMTable component maintains its own active index order, filter and range conditions, current record position, record count statistics, etc.
TDBISAMQuery	The TDBISAMQuery component encapsulates a single SQL statement or multiple SQL statements in DBISAM. These SQL statements may or may not return a result set. It is used to

	search and update data within the physical tables specified by the SQL statement or statements in the SQL property. An SQL statement or statements can be executed by setting the Active property to True, by calling the Open method (for SQL statements that definitely return a result set), or by calling the ExecSQL method (for SQL statements that may or may not return a result set). The DatabaseName property specifies the database where the table or tables reside. Please see the Executing SQL Queries topic for more information. The TDBISAMQuery component descends from the TDBISAMDBDataSet component, which descends from the TDBISAMDataSet component, which descends from the common TDataSet component that is the basis for all data access in Delphi and C++. None of these lower-level components should be used directly and are only for internal structuring purposes in the class hierarchy. You can have multiple TDBISAMQuery components using the same physical table. Each TDBISAMQuery component maintains its own result set filter and range conditions, current record position, record count statistics, etc.

Note
Opening a TDBISAMTable or TDBISAMQuery component will automatically cause its corresponding TDBISAMDatabase component to open, which will also automatically cause its corresponding TDBISAMSession component to start, which will finally cause the TDBISAMEngine to start. This design ensures that the necessary connections for a session, database, etc. are completed before the opening of the table or query is attempted.

2.2 Data Types and NULL Support

Introduction

DBISAM supports the most common data types available for the Delphi and C++ development products as well as the SQL language. Below you will find a listing of the data types with a brief description, their Delphi and C++ equivalent TFieldType type and TField object, and their SQL data type.

> **Note**
> The TFieldType type is also used with the TDBISAMFieldDef, TDBISAMParam, and TDBISAMFunctionParam objects.

Data Type	Description
String	String fields are fixed in length and can store up to 512 characters in a single field. Trailing blank spaces are automatically trimmed from any strings entered into string fields. Internally, String fields are stored as a NULL-terminated string. String fields can be indexed using normal indexes as well as full text indexing. The equivalent Delphi and C++ TFieldType is ftString, the TField object used for String fields is the TStringField object, and the equivalent SQL data type is the VARCHAR type. The SQL VARCHAR data type is specified as: VARCHAR(<number of characters>)
FixedChar	FixedChar fields are basically the same as string fields with the exception that trailing blank spaces are not automatically removed from any strings entered into them. The equivalent Delphi and C++ TFieldType is also ftString, but the TStringField object that represents a FixedChar field will have its FixedChar property set to True. The equivalent SQL data type is either the CHAR or CHARACTER type. The SQL CHAR and CHARACTER data types are specified as: CHAR(<number of characters>) or CHARACTER(<number of characters>)
GUID	GUID fields are basically the same as string fields with the exception that they are fixed at 38 bytes in length and are always used to store the string representation of a GUID value. The equivalent Delphi and C++ TFieldType is ftGuid, the TField object used for GUID fields is the TGuidField object, and the equivalent SQL data type is GUID.
Bytes	Bytes fields are fixed in length and can store up to 512 bytes in a single field. Bytes fields can be indexed using normal indexes only. The equivalent Delphi and C++ TFieldType is ftBytes, the TField object used for Bytes fields is the TBytesField object, and the equivalent SQL data type is BYTES, VARBYTES, BINARY, or VARBINARY. The SQL BYTES, VARBYTES, BINARY, OR VARBINARY data type is specified as:

	BYTES(<number of characters>)
Blob	Blob fields are variable in length and may contain up to 2 gigabytes of data. The data stored in Blob fields is not typed or interpreted in any fashion. Blob fields are stored in a blocked fashion internally in the physical BLOB file that is part of a logical DBISAM table. Blob fields cannot be indexed in any fashion. The equivalent Delphi and C++ TFieldType is ftBlob, the TField object used for Blob fields is the TBlobField object, and the equivalent SQL data type is either the BLOB or LONGVARBINARY type.
Memo	Memo fields are variable in length and may contain up to 2 gigabytes of data minus a NULL terminator. The data stored in Memo fields is always text. Memo fields are stored in a blocked fashion internally in the physical BLOB file that is part of a logical DBISAM table. Memo fields cannot be indexed using normal indexes, but can be indexed using full text indexing. The equivalent Delphi and C++ TFieldType is ftMemo, the TField object used for Memo fields is the TMemoField object, and the equivalent SQL data type is either the MEMO or LONGVARCHAR type.
Graphic	Graphic fields are variable in length and may contain up to 2 gigabytes of data. The data stored in Graphic fields is not typed or interpreted in any fashion, however it is identified in a special way to allow for Delphi and C++ to perform special type-assignments with bitmap and other graphic objects. Graphic fields are stored in a blocked fashion internally in the physical BLOB file that is part of a logical DBISAM table. Graphic fields cannot be indexed in any fashion. The equivalent Delphi and C++ TFieldType is ftGraphic, the TField object used for Graphic fields is the TGraphicField object, and the equivalent SQL data type is the GRAPHIC type.
Date	Date fields contain dates only. Internally, Date fields are stored as a 32-bit integer representing cumulative days. Date fields can be indexed using normal indexes only. The equivalent Delphi and C++ TFieldType is ftDate, the TField object used for Date fields is the TDateField object, and the equivalent SQL data type is DATE.
Time	Time fields contain times only. Internally, Time fields are stored as a 32-bit integer representing cumulative milliseconds. Time fields can be indexed using normal indexes only. The equivalent Delphi and C++ TFieldType is ftTime, the TField object used for Time fields is the TTimeField object, and the equivalent SQL data type is TIME.
TimeStamp	TimeStamp fields contain both a date and a time. Internally, TimeStamp fields are stored as a 64-bit floating-point number (a double) representing cumulative milliseconds. TimeStamp fields can be indexed using normal indexes only. The equivalent Delphi and C++ TFieldType is ftDateTime, the TField object used for TimeStamp fields is the TDateTimeField object, and the equivalent SQL data type is TIMESTAMP.
Boolean	Boolean fields contain logical True/False values. Internally,

	Boolean fields are stored as a 16-bit integer. Boolean fields can be indexed using normal indexes only. The equivalent Delphi and C++ TFieldType is ftBoolean, the TField object used for Boolean fields is the TBooleanField object, and the equivalent SQL data type is BOOLEAN, BOOL, or BIT (compatibility syntax, BOOLEAN or BOOL is preferred).
SmallInt	SmallInt fields contain 16-bit, signed, integers and are stored internally as such. SmallInt fields can be indexed using normal indexes only. The equivalent Delphi and C++ TFieldType is ftSmallInt, the TField object used for SmallInt fields is the TSmallIntField object, and the equivalent SQL data type is SMALLINT.
Word	Word fields contain 16-bit, unsigned, integers and are stored internally as such. Word fields can be indexed using normal indexes only. The equivalent Delphi and C++ TFieldType is ftWord, the TField object used for Word fields is the TWordField object, and the equivalent SQL data type is WORD.
Integer	Integer fields contain 32-bit, signed, integers and are stored internally as such. Integer fields can be indexed using normal indexes only. The equivalent Delphi and C++ TFieldType is ftInteger, the TField object used for Integer fields is the TIntegerField object, and the equivalent SQL data type is INTEGER or INT.
AutoInc	AutoInc fields contain 32-bit, signed, integers and are stored internally as such. AutoInc fields are always editable and you may have more than one AutoInc field per record, with each field incrementing independently. AutoInc fields will increment if you are appending or inserting a record and a value is not specified for the field (field is NULL) when the Post operation occurs, and will leave any existing value alone if one is already specified. AutoInc fields can be indexed using normal indexes only. The equivalent Delphi and C++ TFieldType is ftAutoInc, the TField object used for AutoInc fields is the TAutoIncField object, and the equivalent SQL data type is AUTOINC.
LargeInt	LargeInt fields contain 64-bit, signed, integers and are stored internally as such. LargeInt fields can be indexed using normal indexes only. The equivalent Delphi and C++ TFieldType is ftLargeInt, the TField object used for LargeInt fields is the TLargeIntField object, and the equivalent SQL data type is LARGEINT.
Float	Float fields contain 64-bit floating-point numbers (doubles) and are stored internally as such. Float fields can be indexed using normal indexes only. The equivalent Delphi and C++ TFieldType is ftFloat, the TField object used for Float fields is the TFloatField object, and the equivalent SQL data type is FLOAT.
Currency	Currency fields are the same as Float fields except they are identified in a special way to allow for Delphi and C++ to format their values as monetary values when displayed as strings. The equivalent Delphi and C++ TFieldType is

	ftCurrency, the TField object used for Currency fields is the TCurrencyField object, and the equivalent SQL data type is MONEY. **Note** Don't confuse the Currency field type with the Currency data type found in Delphi and C++. The Currency field type is essentially still a floating-point number and is not always good for storing exact monetary values, whereas the Currency data type is a fixed-point data type that minimizes rounding errors in monetary calculations. If you wish to have accurate financial figures that use up to 4 decimal places stored in DBISAM tables then you should use the BCD data type described next.
BCD	BCD fields contain a 34-byte TBcd type and are stored internally as such. DBISAM always uses a maximum precision of 20 significant digits with BCD numbers, and the maximum scale is 4 decimal places. BCD fields can be indexed using normal indexes only. The equivalent Delphi and C++ TFieldType is ftBCD, the TField object used for BCD fields is the TBCDField object, and the equivalent SQL data type is NUMERIC OR DECIMAL. The SQL NUMERIC or DECIMAL data type is specified as: NUMERIC(<precision>,<scale>)

NULL Support

The rules for NULL support in DBISAM are as follows:

• If a field has not been assigned a value and was not defined as having a default value in the table structure, it is NULL.

• As soon as a field has been assigned a value it is not considered NULL anymore. String, FixedChar, GUID, Blob, Memo, and Graphic fields are an exception this rule. When you assign a NULL value (empty string) to a String, FixedChar, or GUID field the field will be set to NULL. When the contents of a Blob, Memo, or Graphic field are empty, i.e. the length of the data is 0, the field will be set to NULL.

• If the Clear method of a TField object is called the field will be set to NULL.

• NULL values are treated as separate, distinct values when used as an index key. For example, let's say that you have a primary index comprised of one Integer field. If you had a field value of 0 for this Integer field in one record and a NULL value for this Integer field in another record, DBISAM will not report a key violation error. This is a very important point and should be considered when designing your tables. As a general rule of thumb, you should always provide values for fields that are part of the primary index.

• Any SQL or filter expression involving a NULL value and a non-NULL value will result in a NULL result. For example:

```
100.52 * NULL = NULL
```

```
10 + 20 + NULL = NULL
```

The exception to this rule is when concatenating a string value with a NULL. In this case the NULL value is treated like an empty string. For example:

```
'Last Name is ' + NULL = 'Last Name is '
```

Note
String, FixedChar, or GUID field types in DBISAM treat empty strings as equivalent to NULL, and vice-versa, in any filter or SQL expressions.

NULLs with SQL and Filter Operators

The following pseudo-expressions demonstrate the rules regarding NULLs (not empty strings) and the various SQL and filter operators:

Expression	Result
Column = NULL	Returns True if the column is NULL, False, if not
Column <> NULL	Returns True if the column is not NULL, False if it is
Column >= NULL	Returns True if the column is NULL, False if not
Column <= NULL	Returns True if the column is NULL, False if not
Column > NULL	Returns False
Column < NULL	Returns False
Column BETWEEN NULL AND NULL	Returns True if the column is NULL, False if not
Column BETWEEN NULL AND <non-null value>	Returns False
Column BETWEEN <non-null value> AND NULL	Returns False

The rules are slightly different for String, FixedChar, and GUID expressions due to the fact that DBISAM treats empty strings as equivalent to NULL, but also as a valid non-NULL empty string. The following pseudo-expressions demonstrate the rules regarding empty strings and the various SQL and filter operators:

Expression	Result

Column = ''	Returns True if the column is NULL or equal to an empty string, False, if not
Column <> ''	Returns True if the column is not NULL or not equal to an empty string, False if it is
Column >= ''	Returns True if the column is NULL, equal to an empty string, or greater than an empty string, False if not
Column <= ''	Returns True if the column is NULL or equal to an empty string, False if not
Column > ''	Returns True if the column is greater than an empty string
Column < ''	Returns False
Column BETWEEN '' AND ''	Returns True if the column is NULL or equal to an empty string, False if not
Column BETWEEN '' AND <non-empty string>	Returns True if the column is NULL, equal to an empty string, or greater than an empty string, False if not
Column BETWEEN <non-empty string> AND ''	Returns False

Note
The IN and LIKE operators use the same rules as the equivalency (=) operator. The IN operator behaves as if there are a series of equivalency tests joined together by OR operators.

2.3 Exception Handling and Errors

Introduction

One of the first items to address in any application, and especially a database application, is how to anticipate and gracefully handle exceptions. This is true as well with DBISAM. Fortunately, Delphi and C++ both provide elegant exception types and handling. DBISAM uses this exception handling architecture and also expands upon it in several important ways. In certain situations DBISAM will intercept exceptions and trigger events in order to allow for the continuation of a process without the interruption that would occur if the exception were allowed to propagate through the call stack.

DBISAM Exception Types

DBISAM primarily uses the EDBISAMEngineError object as its exception object for all engine errors. This object descends from the EDatabaseError exception object defined in the common DB unit, which itself descends from the common Exception object. This hierarchy is important since it allows you to isolate the type of error that is occurring according to the type of exception object that has been raised, as you will see below when we demonstrate some exception handling.

> **Note**
> DBISAM also raises certain component-level exceptions as an EDatabaseError to maintain consistency with the way the common DB unit and TDataSet component behaves. These mainly pertain to design-time property modifications, but a few can be raised at runtime also.

The EDBISAMEngineError object contains several important properties that can be accessed to discover specific information on the nature of the exception. The ErrorCode property is always populated with a value which indicates the error code for the current exception. Other properties may or may not be populated according to the error code being raised, and a list of all of the error codes raised by the DBISAM engine along with this information can be found in Appendix B - Error Codes and Messages.

Exception Handling

The most basic form of exception handling is to use the try..except block (Delphi) or try..catch (C++) to locally trap for specific error conditions. The error code that is returned when an open fails due to access problem is 11013, which is defined as DBISAM_OSEACCES in the dbisamcn unit (Delphi) or dbisamcn header file (C++). The following example shows how to trap for such an exception on open and display an appropriate error message to the user:

```
{
      {
      MyDBISAMTable->DatabaseName="c:\testdata";
      MyDBISAMTable->TableName="customer";
      MyDBISAMTable->Exclusive=true;
      MyDBISAMTable->ReadOnly=False;
      try
         {
         MyDBISAMTable->Open();
         }
      catch(const Exception &E)
         {
```

```
          if (dynamic_cast<EDatabaseError*>(E) &
              dynamic_cast<EDBISAMEngineError*>(E))
             {
             if (dynamic_cast<EDBISAMEngineError&>(*E)->ErrorCode==
                 DBISAM_OSEACCES)
                {
                ShowMessage("Cannot open table "+TableName+
                     ", another user has the table open already");
                }
             else
                {
                ShowMessage("Unknown or unexpected "+
                     "database engine error # +IntToStr(
                     dynamic_cast<EDBISAMEngineError&>(*E)->ErrorCode));
                }
             }
          else
             {
             ShowMessage("Unknown or unexpected "+
                     "error has occurred");
             }
          }
       }
}
```

Exception Events

Besides trapping exceptions with a try..except or try..catch block, you may also use a global TApplication::OnException event handler to trap database exceptions. However, doing so will eliminate the ability to locally recover from the exception and possibly retry the operation or take some other course of action. There are several events in DBISAM components that allow you to code event handlers that remove the necessity of coding try..except or try..catch blocks while still providing for local recovery. These events are as follows:

Event	Description
OnEditError	This event is triggered when an error occurs during a call to the TDBISAMTable or TDBISAMQuery Edit method . The usual cause of an error is a record lock failure if the current session is using the pessimistic locking protocol (the default). Please see the Updating Tables and the Locking and Concurrency topics for more information on using this event and the DBISAM locking protocols.
OnDeleteError	This event is triggered when an error occurs during a call to the TDBISAMTable or TDBISAMQuery Delete method. The usual cause of an error is a record lock failure (a record lock is always obtained before a delete regardless of the locking protocol in use for the current session). Please see the Updating Tables and Query Result Sets and the Locking and Concurrency topics for more information on using this event and the DBISAM locking protocols.
OnPostError	This event is triggered when an error occurs during a call to the TDBISAMTable or TDBISAMQuery Post method. The usual cause of an error is a key violation for a unique index or the violation of a table constraint, however it can also be

	triggered by a record lock failure if the locking protocol for the current session is set to optimistic. Please see the Updating Tables and the Locking and Concurrency topics for more information on using this event and the DBISAM locking protocols.
OnQueryError	This event is triggered when an error occurs during the preparation or execution of an SQL statement or script via the TDBISAMQuery ExecSQL or Open methods. If this event is assigned an event handler then it will get triggered and the event handler will have the option of aborting the current SQL statement or script. If this event is not assigned an event handler then this event will not be triggered and the exception will be raised.
OnDataLost	This event is triggered when an error occurs during the alteration of a table's structure via the TDBISAMTable AlterTable or AddIndex methods, or via the execution of the ALTER TABLE or CREATE INDEX SQL statements by the TDBISAMQuery ExecSQL method. An error can be caused by key violations, field deletions, field conversion problems, table constraint failures, and any other type of problem during these operations. The OnDataLost event allows you to react to these errors by cancelling the current operation, continuing, or continuing without triggering this event anymore.

The above events are all based in the TDBISAMTable or TDBISAMQuery components, and are mainly geared toward application-level exception handling. There is a lower level of exception handling available also in the following TDBISAMEngine events:

Event	Description
OnInsertError	This event is triggered whenever an exception occurs during the insertion of any record in any table. The event handler for this event can choose to retry, abort, or fail the insert operation that raised the exception.
OnUpdateError	This event is triggered whenever an exception occurs during the update of any record in any table. The event handler for this event can choose to retry, abort, or fail the update operation that raised the exception.
OnDeleteError	This event is triggered whenever an exception occurs during the deletion of any record in any table. The event handler for this event can choose to retry, abort, or fail the delete operation that raised the exception.

Note
If any exception is raised in an BeforeInsertTrigger, AfterInsertTrigger, BeforeUpdateTrigger, AfterUpdateTrigger, BeforeDeleteTrigger, or AfterDeleteTrigger event, the exception will be converted into an EDBISAMEngineError exception object with an error code of DBISAM_TRIGGERERROR. The original exception's error message will be assigned to the ErrorMessage property of the EDBISAMEngineError exception object, as well as be included as part of the error message in the EDBISAMEngineError exception object itself.

2.4 Configuring and Starting the Server

Introduction

There are no extra steps required in order to use the TDBISAMEngine component in DBISAM as a client engine since the default value of the EngineType property is etClient. However, in order to use the TDBISAMEngine component in DBISAM as a database server you will need to make some property changes before starting the engine.

Configuration Properties

The TDBISAMEngine component has several key properties that are used to configure the database server, which are described below in order of importance:

Property	Description
EngineType	In order to start the TDBISAMEngine component as a database server, you must set this property to etServer.
EngineSignature	Normally this property is left at the default value. However, if you do choose to change this property, you must make sure that it is set to desired value before starting the server. The default value is "DBISAM_SIG". Please see the Customizing the Engine topic for more information.
ServerName	This property is used to identify the database server to external clients once they have connected to the database server. The default value is "DBSRVR".
ServerDescription	This property is used in conjunction with the ServerName property to give more information about the database server to external clients once they have connected to the database server. The default value is "DBISAM Database Server".
ServerMainAddress	This property specifies the IP address that the database server should bind to when listening for regular incoming data connections. The default value is blank (""), which specifies that the database server should bind to all available IP addresses.
ServerMainPort	This property specifies the port that the database server should bind to when listening for regular incoming data connections. The default value is 12005.
ServerMainThreadCacheSize	This property specifies the number of threads that the database server should actively cache for regular data connections. When a thread is terminated on the server it will be added to this thread cache until the number of threads cached reaches this property value. This allows the database server to re-use the threads from the cache instead of having to constantly create/destroy the threads as needed, which can improve the performance of the database server if there are many connections and disconnections occurring. The default value is 10.

ServerAdminAddress	This property specifies the IP address that the database server should bind to when listening for incoming administrative connections. The default value is blank (""), which specifies that the database server should bind to all available IP addresses.
ServerAdminPort	This property specifies the port that the database server should bind to when listening for incoming administrative connections. The default value is 12006.
ServerAdminThreadCacheSize	This property specifies the number of threads that the database server should actively cache for administrative connections. The default value is 1.
ServerEncryptedOnly	This property specifies whether all incoming regular data connections should be encrypted or not. If this property is set to True, then all incoming regular data connections to the database server that are not encrypted will be rejected with the error code 11277, which is defined as DBISAM_REMOTEENCRYPTREQ in the dbisamcn unit (Delphi) or dbisamcn header file (C++). The default value is False. **Note** Administrative connections to the database server must always encrypted and will be rejected if they are not encrypted, regardless of the current value of this property.
ServerEncryptionPassword	This property specifies the password to use for all encrypted connections. If an incoming encrypted connection does not use a password that matches this value of this property, the database server will return the error code 11308, which is defined as DBISAM_REMOTEINVREQUEST in the dbisamcn unit (Delphi) or dbisamcn header file (C++), when any call to the database server is attempted after the connection is made. The default value is "elevatesoft". **Note** If you intend to use encrypted connections to a database server over a public network then you should always use a different encryption password from the default password.
ServerConfigFileName	This property specifies the name of the configuration file that the database server will use for storing all server configuration information including users, databases, server-side procedures, user rights, and scheduled events. This file is compressed and encrypted, and a backup is made, with the extension ".scb", any time a modification is made. The default value is "dbsrvr.scf".

	Note Any new configuration file name specified via this property will be given the default extension of ".scf" automatically.
ServerConfigPassword	This property specifies the password to use to encrypt the contents of the server configuration file. This ensures that if someone does obtain physical access to the configuration file that they will still be unable to read its contents, especially user names and passwords, without this password.

Note
All of the properties of the TDBISAMEngine component described above can only be modified when the Active property is False and the engine has not been started.

Starting the Server

Once you have configured the database server using the above properties, starting the server is quite simple. All you need to do is set the Active property to True. The following shows an example of how one might configure and start a database server using the default global Engine function in the dbisamtb unit (Delphi) or dbisamtb header file (C++):

```
{
   Engine()->ServerName="MyTestServer";
   Engine()->ServerDescription="My Test Server";
   // Only listen on this IP address
   Engine()->ServerMainAddress="192.168.0.1";
   Engine()->ServerConfigFileName="mytest.scf";
   Engine()->ServerConfigPassword="test123456";
   Engine()->Active=true;
}
```

Note
You can also use the TDBISAMEngine OnStartup event to configure the TDBISAMEngine component before it is started.

Default Login Information

The default user ID and password for the database server are:

User ID: Admin (case-insensitive)
Password: DBAdmin (case-sensitive)

This user has full administrator privileges and is widely known, so it is recommended that you delete it as a user once you have established another administrative user on the database server.

Database Servers Provided with DBISAM

DBISAM comes with an application (GUI) database server project for Delphi called dbsrvr.dpr and a command-line (console) database server project for Delphi called dbcmd.dpr. You can examine the source code of these projects to see how you would go about setting up a TDBISAMEngine component as a database server in a project. Both of these projects are also provided in compiled form with DBISAM.

> **Note**
> If you wish to run either database server, either as a normal application or a Windows service, you must copy the desired database server executable into a directory with read/write permissions, based upon the user account under which you wish to run the database server, and run it from that directory. This requirement is due to the fact that the database server writes its log and configuration files to the directory where the database server executable is located. This is a legacy behavior that is not compatible with running the database server from the default installation directory in the default \Program Files (x86) sub-tree.

The dbsrvr database server can be run as a normal application or as a Windows service. When running the dbsrvr database as a normal application, the server will display an icon in the system tray that can be right-clicked to obtain general information about the server, as well as start and stop the server. You can find the dbsrvr database server in the \servers\dbsrvr sub-directory under the main installation directory for the version of DBISAM that you installed.

If you wish to run the dbsrvr database server as a Windows service you must first install it as a service by running the database server with the /install command-line switch set. For example, to install the database server as a service using a command prompt window under Windows (2000 or higher) you would specify the following command:

```
dbsrvr.exe /install
```

To uninstall the dbsrvr database server as a Windows service you must run the database server with the /uninstall command-line switch set. For example, to uninstall the dbsrvr database server as a service using a command-prompt window under Windows (2000 or higher) you would specify the following command:

```
dbsrvr.exe /uninstall
```

> **Note**
> You must run the above commands while logged in as an Administrator, or they will not succeed and you will see an "Access Denied" error message. Also, the dbsrvr database server will not display an icon in the system tray, nor will it display a user interface, when run as a Windows service. Recent versions of Windows restrict services from interacting with the desktop in order to permit them to run in non-GUI server environments.

After installing the dbsrvr database server as a Windows service, you can run the database server by starting the service interactively via the Windows Services management console, or by using the net start command-line command:

```
net start dbsrvr
```

You can stop the database service interactively via the Windows Services management console, or by using the net stop command-line command:

```
net stop dbsrvr
```

The dbcmd database server can only be run as a normal (console) application. You can find the dbcmd database server in the \servers\dbcmd sub-directory under the main installation directory for the version of DBISAM that you installed.

The database servers will accept commmand-line switches that affect their behavior. The following switches are supported when starting up either database server:

Switch	Description
/sn	Server name parameter The /sn switch specifies the user-defined server name that will be used to identify the server to remote sessions. You must enclose the server name in double quotes if there are spaces in the server name. The server name is informational only.
/sd	Server description parameter The /sd switch specifies the user-defined server description that will be displayed in the caption of the server's user interface. You must enclose the server description in double quotes if there are spaces in the server description. The server description is informational only.
/sa	Server address parameter The /sa switch specifies the main server IP address that the server will bind to for accepting inbound data connections. The IP address must be specified directly after the /sp switch using dot notation (i.e. 192.168.0.1). The default IP address that the server will bind to if this switch is not specified is all IP addresses available on the machine. Using this option will cause the server to only listen on the specified address. This means that it will no longer listen on the local loopback 127.0.0.1 address.
/sp	Server port parameter The /sp switch specifies the main server port that the server will bind to for accepting inbound data connections. The port number must be specified directly after the /sp switch. The default main port that the server will bind to if this switch is not specified is 12005.
/st	Server thread cache size parameter The /st switch specifies the main server thread cache size.

	The thread cache size controls how many threads the server will cache in order to speed up connect/disconnect times. The thread cache size must be specified directly after the /st switch. The default main thread cache size that the server will use if this switch is not specified is 10.
/aa	Administration address parameter The /aa switch specifies the administration server IP address that the server will bind to for accepting administrative connections. The IP address must be specified directly after the /aa switch using dot notation (i.e. 192.168.0.1). The default administration IP address that the server will bind to if this switch is not specified is all IP addresses available on the machine. Using this option will cause the server to only listen on the specified address. This means that it will no longer listen on the local loopback 127.0.0.1 address.
/ap	Administration port parameter The /ap switch specifies the administration server port that the server will bind to for accepting administrative connections. The port number must be specified directly after the /ap switch. The default administration port that the server will bind to if this switch is not specified is 12006.
/at	Administration thread cache size parameter The /at switch specifies the administration server thread cache size. The thread cache size controls how many threads the server will cache in order to speed up connect/disconnect times. The thread cache size must be specified directly after the /at switch. The default administration thread cache size that the server will use if this switch is not specified is 1.
/cf	Configuration file name parameter The /cf switch specifies the server configuration file name. The configuration file is where the server stores all configuration information including databases, users, permissions, etc. You must enclose the configuration file name in double quotes if there are spaces in the configuration file name. Do not specify a file extension for the file since the server always uses the ".scf" extension for all configuration files. The default configuration file name that the server will use if this switch is not specified is "dbsrvr".
/cp	Configuration file password parameter The /cp switch specifies the server configuration file password. The configuration file password is used to encrypt the contents of the configuration file. You must enclose the configuration file password in double quotes if there are spaces in the configuration file password. The default configuration file password that the server will use if this switch is not specified is "elevatesoft".

	Note Do not lose this password. If you do the server will not be able to read the configuration information and there is no way for Elevate Software to retrieve the configuration information.
/en	Encrypted connections only parameter The /en switch specifies that the main server will require encrypted connections only. By default the administration server always requires encypted connections, but normally encrypted connections are not required for the main server.
/ep	Encrypted connnection password parameter The /ep switch specifies the password to use for encrypting all data between any remote sessions and the main and administration server. This switch can be specified without the above /en switch to change the password for encrypted connections to the administration server only. If combined with the above switch, this switch will change the password for encrypted connections to both the main server and the administration server. You must enclose the encryption password in double quotes if there are spaces in the encryption password. The default encryption password that the server will use if this switch is not specified is "elevatesoft". **Note** If this password is not set to the same value that is used by the remote sessions connecting to either the main or administration server, the remote sessions will receive errors and not be able to connect to the server at all.
/al	Append to log parameter The /al switch specifies that the server should append to any existing server log file when the server process is started. The default behavior is to overwrite the log every time the server process is started.

The only difference between starting the dbsrvr database server as a normal application and starting the dbsrvr database server as a Windows service is in the way the switches are specified. When the dbsrvr database server is started as a normal application, you may specify the switches directly on the command-line that you are using to start the database server. For example the following command will start the dbsrvr database server using port 13000 for the main port and 13001 for the administration port:

```
dbsrvr.exe /sp13000 /ap13001
```

When the dbsrvr database server is started as a Windows service, you may specify the switches via the Startup Parameters in the properties for the service in the Services management console, or directly on the command-line that you are using to start the dbsrvr database server with the net start command. For example the following command will start the database server as a service with it using port 13000 for the main port and 13001 for the administration port:

```
net start dbsrvr /sp13000 /ap13001
```

Note
In order to start the dbsrvr database server as a Windows service the database server must have already been installed as a service using the /install command-line switch.

In addition to using command-line parameters, you may also use an .ini file to specify the parameters for the database server. The following is a sample .ini file that can be used with either the dbsrvr or dbcmd database servers:

```
; Sample DBISAM Database Server Parameters INI File

[Server Parameters]
; Default server name is the EXE name
Server Name=Test Server
; Default server description is DBISAM Database Server
; plus the Server Name
Server Description=Test Server Description
; Default server IP address is all addresses on the machine
Server Address=127.0.0.1
; Default server port is 12005
Server Port=10001
; Default server thread cache size is 10
Server Thread Cache Size=20
; Default admin IP address is all addresses on the machine
Administration Address=127.0.0.1
; Default admin port is 12006
Administration Port=10002
; Default admin thread cache size is 1
Administration Thread Cache Size=4
; Default configuration file name is dbsrvr
Configuration File=Test
; Default configuration file password is elevatesoft
Configuration Password=cannotguessme
; 0=main server allows unencrypted connections (default)
; 1=main server allows only encrypted
Encrypted Only=0
; Default encryption password is elevatesoft
Encryption Password=cannotguessme
; 0=overwrite log file (default) 1=append to log file
Append To Log=0

; SQL performance logging

; 0=no SQL performance logging (default) 1=log all statements with execution
      times above the min execution time (below)
SQL Performance Tracking=0
```

```
; Minimum execution time, in seconds, required before an SQL statement is
      logged (default is 30 seconds)
Min SQL Performance Execution Time=30
; SQL performance log file name
SQL Performance File Name=
; Max SQL performance log file size (default is 128MB)
Max SQL Performance File Size=134217728
; 0=no auto-incrementing of SQL performance log file name 1=auto-increment
      SQL performance log file name
Auto-Increment SQL Performance File Name=0
; Max SQL performance log file autoinc (default is 64)
Max Auto-Increment SQL Performance File Name=64
```

Note
The .ini file that contains the database server parameters must have the same root file name as the database server itself. For example, if you wanted to use the above .ini file with the dbsrvr database server, you would need to save it to a dbsrvr.ini file in the same directory as the dbsrvr.exe executable in order for the database server to find it and use its contents. Likewise, if you wanted to use it with the dbcmd database server, you would need to save it to a dbcmd.ini file in the same directory as the dbcmd.exe executable.

Multiple instances of the database server can be started on the same physical machine. The root name of the database server executable is used to determine the name of the log and parameters (.ini) files to use when the database server is started. Also, when running as a Windows service the dbsrvr database server relies on the root name of the database server executable to determine the service name. What this means is that to have multiple instances of the database server running on the same machine you must put them in separate physical directories if running them as a normal application, or copy the database server to different executable files with a different root name if running them as a service. For example, if you wanted to run two instances of the database server as services, you would copy the dbsrvr.exe to two separate executable files called dbsrvr1.exe and dbsrvr2.exe. Then you would install and run them as follows:

First database server

```
dbsrvr1.exe /install

net start dbsrvr1
```

Second database server

```
dbsrvr2.exe /install

net start dbsrvr2
```

2.5 Server Administration

Introduction

Administering a database server involves maintaining global connection settings as well as databases, server-side procedures, users, and scheduled events. All of this information is stored in the configuration file specified via the TDBISAMEngine ServerConfigFileName property. DBISAM offers the ability to administer a database server both locally through the TDBISAMEngine component and remotely through the TDBISAMSession component. Both types of administration are very similar except for some minor differences.

Local Administration

Local administration of a database server involves calling methods of the TDBISAMEngine component directly. The following methods are all for local administrative use:

Method	Description
GetServerConfig	This method retrieves the global database server settings for maximum allowed connections, connection timeouts, dead session settings, the temporary files directory, and authorized and blocked IP addresses.
ModifyServerConfig	This method modifies the global database server settings.
GetServerLogCount	This method retrieves the total number of log records present in the current log file. Calling this method triggers the TDBISAMEngine OnServerLogCount event. If an event handler is defined for this event, then it is called to retrieve the count from whatever storage medium is being used for the log file.
GetServerLogRecord	This method retrieves the Nth log record from the current log file. Calling this method triggers the TDBISAMEngine OnServerLogRecord event. If an event handler is defined for this event, then it is called to retrieve the specified log record from whatever storage medium is being used for the log file.
StartAdminServer	This method causes the database server to begin listening for administrative connections on the IP address and port specified by the TDBISAMEngine ServerAdminAddress and ServerAdminPort properties.
StopAdminServer	This method causes the database server to stop listening for administrative connections.
StartMainServer	This method causes the database server to begin listening for regular data connections on the IP address and port specified by the TDBISAMEngine ServerMainAddress and ServerMainPort properties. Calling this method triggers the TDBISAMEngine OnServerStart event.
StopMainServer	This method causes the database server to stop listening for regular data connections. Calling this method triggers the TDBISAMEngine OnServerStop event.

GetServerUTCDateTime	This method retrieves the current date and time from the server in UTC (Coordinated Universal Time).
GetServerUpTime	This method retrieves the total up time of the database server in seconds.
GetServerMemoryUsage	This method has been deprecated and always returns 0 as of version 4.17 of DBISAM and the introduction of the new memory manager used in the DBISAM database server. Please see the Replacement Memory Manager topic for more information.
GetServerSessionCount	This method retrieves the total number of sessions present on the database server at the time of the method call. **Note** This count does not include administrative sessions, only regular sessions.
GetServerConnectedSessionCount	This method retrieves the total number of sessions on the database server that are currently connected at the time of the method call. **Note** This count does not include administrative sessions, only regular sessions.
GetServerSessionInfo	This method retrieves information about the specified session such as its unique session ID, when it was created, when it was last connected, the user name, the IP address of the user, and whether the connection is encrypted. **Note** This method does not return information about administrative sessions, only regular sessions.
DisconnectServerSession	This method disconnects the specified session, but does not remove the session from the database server. Once the session is disconnected it is considered to be a dead session. **Note** This method cannot be used on administrative sessions, only regular sessions.
RemoveServerSession	This method removes the specified session completely from the database server.

	Note This method cannot be used on administrative sessions, only regular sessions.
GetServerUserNames	This method will retrieve a list of user names that are currently defined on the database server.
GetServerUser	This method will retrieve information about a specific user, including the user's password, a description of the user, and whether the user is an administrator for this server.
AddServerUser	This method will add a new user.
ModifyServerUser	This method will modify a user's information.
ModifyServerUserPassword	This method will modify only a user's password.
DeleteServerUser	This method will delete a user.
GetServerDatabaseNames	This method will retrieve a list of database names that are currently defined on the database server.
GetServerDatabase	This method will retrieve information about a specific database, including the database's description and the actual physical path to the database tables. **Note** All database server physical path information for databases defined on the server are interpreted relative to the drives, directories, etc. available to the database server.
AddServerDatabase	This method will add a new database.
ModifyServerDatabase	This method will modify a database's information.
DeleteServerDatabase	This method will delete a database.
GetServerDatabaseUserNames	This method will retrieve a list of users that are assigned rights to a specific database.
GetServerDatabaseUser	This method will retrieve the user rights of a user for a specific database.
AddServerDatabaseUser	This method will add user rights for a specific user to a specific database.
ModifyServerDatabaseUser	This method will modify the user rights of a user for a specific database.
DeleteServerDatabaseUser	This method will delete the user rights of a user for a specific database.
GetServerProcedureNames	This method will retrieve a list of server-side procedure names that are currently defined on the database server.
GetServerProcedure	This method will retrieve information about a specific server-side procedure, specifically the server-side procedure's description.

AddServerProcedure	This method will add a new server-side procedure.
ModifyServerProcedure	This method will modify a server-side procedure's information.
DeleteServerProcedure	This method will delete a server-side procedure.
GetServerProcedureUserNames	This method will retrieve a list of users that are assigned rights to a specific server-side procedure.
GetServerProcedureUser	This method will retrieve the user rights of a user for a specific server-side procedure.
AddServerProcedureUser	This method will add user rights for a specific user to a specific server-side procedure.
ModifyServerProcedureUser	This method will modify the user rights of a user for a specific server-side procedure.
DeleteServerProcedureUser	This method will delete the user rights of a user for a specific server-side procedure.
GetServerEventNames	This method will retrieve a list of scheduled event names that are currently defined on the database server.
GetServerEvent	This method will retrieve information about a specific scheduled event, specifically the scheduled event's description and scheduling parameters.
AddServerEvent	This method will add a new scheduled event.
ModifyServerEvent	This method will modify a scheduled event's information.
DeleteServerEvent	This method will delete a scheduled event.

Remote Administration

Remotely administering a database server involves connecting to the server's administration port using a TDBISAMSession component that has the following properties set properly:

Property	Setting
SessionType	This property must be set to stRemote.
RemoteEncryption	This property must be set to True.
RemoteEncryptionPassword	This property must be set to the same password as the ServerEncryptionPassword property of the TDBISAMEngine component that the session is connecting to.
RemoteAddress	This property must be set to the IP address of the database server as it appears to remote machines. You may optionally use the RemoteHost property if there is DNS information available for the database server that you are connecting to.
RemotePort	This property must be set to the administrative port as it appears to remote machines. You may optionally use the RemoteService property if there is service information available for the database server administrative port that you are connecting to.

	Note There is an important distinction to make here. The IP address and port specified for a remote session is not always the same as the IP address and port specified in the ServerAdminAddress and ServerAdminPort properties of the TDBISAMEngine component that the session is connecting to. This is because network routers can use port forwarding and other techniques to forward network traffic destined for a specific public IP address and port to a private, internal LAN IP address and port.
RemoteUser	This property must be set to the name of a valid administrator user for the database server that you are connecting to.
RemotePassword	This property must be set to the proper password for the user name specified by the RemoteUser property.

Once you have set up the TDBISAMSession properties properly for administrative access you can proceed to call the TDBISAMSession Open method or set the TDBISAMSession Active property to True. This will cause the remote session to attempt to connect to the database server on the administrative port. Provided that you have set up everything properly, you will connect to the database server on the administrative port and can then proceed to use the remote administrative methods of the TDBISAMSession component to administer the database server.

The remote administration methods of the TDBISAMSession component are identical to the local methods of the TDBISAMEngine component except that the TDBISAMSession methods are named *Remote* instead of *Server*. Also, there are no StartAdminServer or StopAdminServer methods, and the StartMainServer and StopMainServer methods are called StartRemoteServer and StopRemoteServer, respectively. For a complete list of the remote administration methods please see the TDBISAMSession component.

The following example shows how to set up the TDBISAMSession properties for remotely administering a database server, connect to the database server, add a new user (not an administrator), and then add user rights to the "AccountingDB" database for this user:

```
{
   MyDBISAMSession->SessionType=stRemote;
   MyDBISAMSession->RemoteEncryption=true;
   // Assume the default encryption password in use
   MyDBISAMSession->RemoteAddress="192.168.0.1";
   MyDBISAMSession->RemotePort=12006;
   MyDBISAMSession->RemoteUser="Admin";
   MyDBISAMSession->RemotePassword="DBAdmin";
   MyDBISAMSession->Open();
   try
      {
      MyDBISAMSession->AddRemoteUser("Test","Test123456",
                                     "Test User",false);
      MyDBISAMSession->AddRemoteDatabaseUser("AccountingDB","Test",
            TDatabaseRights()<<drRead<<drInsert<<drUpdate<<drDelete);
      }
   __finally
      {
      MyDBISAMSession->Close();
```

```
        }
}
```

Note
The Server Administration Utility that can be found in the additional software download (DBISAM-ADD) on the Elevate Software web site also comes complete with source code and demonstrates how to use all of the remote administration functionality described above.

2.6 Customizing the Engine

Introduction

As already discussed in the DBISAM Architecture topic, the TDBISAMEngine component represents the engine in DBISAM. The following information will show how to customize the engine in an application. Some of the customizations can be made for the engine when it is acting as a local engine or server engine, while other customizations are only intended for the server engine. The TDBISAMEngine EngineType property controls whether the engine is behaving as a local engine or a server engine.

Engine Signature

The TDBISAMEngine EngineSignature property controls the engine signature for the engine. The default engine signature is "'DBISAM_SIG". The engine signature in DBISAM is used to "stamp" all tables, backup files, and streams created by the engine so that only an engine with the same signature can open them or access them afterwards. If an engine does attempt to access an existing table, backup file, or stream with a different signature than that of the table, backup file, stream, an EDBISAMEngineError exception will be raised. The error code that is returned when the access fails due to an invalid engine signature is 12036 and is defined as DBISAM_BADSIGNATURE in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

Also, if the EngineType property is set to etClient, the engine signature is used to stamp all requests sent from a remote session to a database server. If the database server is not using the same engine signature then the requests will be treated as invalid and rejected by the database server. If the EngineType property is set to etServer, the engine signature is used to stamp all responses sent from the database server to any remote session. If the remote session is not using the same engine signature then the requests will be treated as invalid and rejected by the database server. In summary, both the remote sessions and the database server must be using the same engine signature or else communications between the two will be impossible.

Triggers

Triggers can be implemented for a local or server engine by using the TDBISAMEngine StartTransactionTrigger, CommitTrigger, RollbackTrigger, BeforeInsertTrigger, AfterInsertTrigger, BeforeUpdateTrigger, AfterUpdateTrigger, BeforeDeleteTrigger, AfterDeleteTrigger, RecordLockTrigger, and RecordUnlockTrigger events. These events are fired whenever a transaction is started, committed, or rolled back, and whenever a record is inserted, updated, deleted, locked, or unlocked via navigational methods or via SQL. However, these events are not triggered during any system processing such as Creating and Altering Tables or Optimizing Tables. This allows for the freedom to change the table metadata without having to worry about causing any errors due to constraints that may be enforced via the triggers.

> **Note**
> These events can be called from multiple threads concurrently, so it is very important that you observe the rules of multi-threading with DBISAM. The TDBISAMSession and TDBISAMDatabase components are created automatically by the engine and passed as parameters to these events, so if you create any TDBISAMTable or TDBISAMQuery components in an event handler for one or more of these events, you need to make sure to assign the SessionName and DatabaseName properties to that of these passed TDBISAMSession and TDBISAMDatabase components. Please see the Multi-Threaded Applications topic for more information.

The TDBISAMEngine triggers events can be used for audit logging, referential integrity, replication, hot backups, etc. There really is no limit to what can be coded in an event handler attached to one or more of these events. The following is an example of a BeforeDelete trigger that executes a query in order to determine whether to permit the deletion or raise an exception:

```
void __fastcall TMyForm::EngineBeforeDeleteTrigger(TObject *Sender,
      TDBISAMSession *TriggerSession, TDBISAMDatabase *TriggerDatabase,
      const AnsiString TableName, TDBISAMRecord *CurrentRecord)
{
   TDBISAMQuery *OrdersQuery;
   if (AnsiCompareText(TableName,"customer")==0)
      {
      TDBISAMQuery *OrdersQuery=TDBISAMQuery->Create(nil);
      try
         {
         OrdersQuery->SessionName=TriggerDatabase->SessionName;
         OrdersQuery->DatabaseName=TriggerDatabase->DatabaseName;
         OrdersQuery->RequestLive=true;
         OrdersQuery->SQL->Text="SELECT * FROM Orders "+
                     "WHERE CustNo=:CustNo AND "+
                     "AmountPaid < ItemsTotal";
         OrdersQuery->ParamByName("CustNo")->AsFloat=
                   CurrentRecord->FieldByName("CustNo")->AsFloat;
         OrdersQuery->Open();
         try
            if(RecordCount > 0)
               {
               throw Exception("Cannot delete this "+
                              "customer, there are still "+
                              IntToStr(RecordCount)+' active "+
                              "orders present for this "+
                              "customer");
               }
         __finally
            {
            OrdersQuery->Close();
            }
         }
      __finally
         {
         delete OrdersQuery;
         }
      }
}
```

You can use the TDBISAMEngine OnInsertError, OnUpdateError, and OnDeleteError events to trap any errors that may occur during an insert, update, or delete, and reverse any action that may have been initiated in a Before*Trigger event handler. For example, if you start a transaction in a BeforeDeleteTrigger event, you should be sure to rollback the transaction in an OnDeleteError event handler or else you will inadvertently leave an active transaction hanging around.

The TriggerSession CurrentServerUser property can be referenced from within a trigger that is being executed when the TDBISAMEngine EngineType property is set to etServer in order to retrieve the current user name.

Note
If any exception is raised in any trigger event handler, the exception will be converted into an EDBISAMEngineError exception object with an error code of DBISAM_TRIGGERERROR. The original exception's error message will be assigned to the ErrorMessage property of the EDBISAMEngineError exception object, as well as be included as part of the error message in the EDBISAMEngineError exception object itself.

Custom SQL and Filter Functions

Custom SQL and filter functions can be implemented for a local or server engine by using the TDBISAMEngine Functions property in conjunction with the OnCustomFunction event. The Functions property is a TDBISAMFunctions object, and the easiest way to add new functions is to use the Functions property's CreateFunction method, which will create a new TDBISAMFunction object, add it to the Functions property, and return a reference to the new function. You can then use this function reference to add the parameters to the function using the TDBISAMFunction Params property. The Params property is a TDBISAMFunctionParams object, and the easiest way to add new function parameters is to use the Params property's CreateFunctionParam method, which will create a new TDBISAMFunctionParam object, add it to the Params property, and return a reference to the new function parameter. You can then use this function parameter reference to specify the data type of the parameters to the custom function. All custom function result and parameter data types use the TFieldType type. Please see the Data Types and NULL Support topic for more information.

The following example shows how you would use the CreateFunction method to create a function called "DaysBetween" that returns the number of days between two date parameters as an integer:

```
{
   // We'll just use the default Engine global function
   // for this example
   TDBISAMFunction *NewFunction=Engine()->Functions->
          CreateFunction(ftInteger,"DaysBetween");
   NewFunction->CreateFunctionParam(ftDate);
   NewFunction->CreateFunctionParam(ftDate);
}
```

Note
Adding a custom function while the engine is active will result in the engine triggering an exception. You should define all custom functions before activating the engine.

Once you have defined the custom function using the TDBISAMEngine Functions property, you must then proceed to implement the function using an event handler assigned to the TDBISAMEngine OnCustomFunction event. When DBISAM encounters a function name in a filter or SQL expression that does not match that of a pre-defined function in DBISAM, the OnCustomFunction event is triggered with the name of the function, the parameters to the function defined as a TDBISAMParams object, and a parameter for returning the function result as a variant variable. Inside of the OnCustomFunction event handler you must conditionally process each function using the name of the function passed to the event handler. The following example implements the "DaysBetween" function that we defined previously in the above example:

```
void __fastcall TMyForm::CustomFunction(TObject *Sender,
      const AnsiString FunctionName, TDBISAMParams *FunctionParams,
      Variant &Result)
{
   if (AnsiCompareText(FunctionName,"DaysBetween")==0)
      {
      // Notice that the function parameters are accessed
      // in a 0-based manner
      TTimeStamp Stamp1=DateTimeToTimeStamp(
            FunctionParams->Items[0]->AsDateTime);
      TTimeStamp Stamp2=DateTimeToTimeStamp(
            FunctionParams->Items[1]->AsDateTime);
      Result=Trunc((Stamp2.Date-Stamp1.Date)+
                  (((((Stamp2.Time-Stamp1.Time)/1000)/60)/60)/24));
      }
}
```

Note
The name of the parameters sent to the OnCustomFunction event handler will be:

"Param" plus an underscore (_) plus the position of the parameter (0-based)

for constants or expressions, and:

Table name plus underscore (_) plus column name plus (_) plus the position of the parameter (0-based)

for table columns. This allows you to identify which column from which table was passed to a custom function.

Memory Buffer Customizations

The TDBISAMEngine MaxTableDataBufferCount, MaxTableDataBufferSize, MaxTableIndexBufferCount, MaxTableIndexBufferSize, MaxTableBlobBufferCount, and MaxTableBlobBufferSize properties allow you to control how much memory is used for buffering the data records, index pages, and BLOB blocks for each physical table opened in a given session in the engine. The *Size properties dictate how much memory, in bytes, to allocate. The *Count properties dictate the maximum number of data records, index pages, and BLOB blocks that can be allocated regardless of the amount of memory available. This is to ensure that the buffering architecture in DBISAM does not get overwhelmed by buffering too many small records, etc.

Lock File Name Customizations

The default lock file name, "dbisam.lck", can be modified using the TDBISAMEngine LockFileName property.

File Extension Customizations

The default file extensions for tables are detailed in the DBISAM Architecture topic. You can modify these default extensions using the following properties:

Extensions	Properties

Tables	TableDataExtension TableIndexExtension TableBlobExtension
Backup Files	TableDataBackupExtension TableIndexBackupExtension TableBlobBackupExtension
Upgrade Backup Files	TableDataUpgradeExtension TableIndexUpgradeExtension TableBlobUpgradeExtension
Temporary Files	TableDataTempExtension TableIndexTempExtension TableBlobTempExtension

Note
The temporary file extension customizations are useful when you wish to have any temporary tables created by DBISAM use a file extension other than .dat, .idx, or .blb. Recent issues with certain anti-virus software has shown that it may be necessary to change the extensions of the files that make up temporary DBISAM tables in order to prevent the anti-virus software from interfering with DBISAM's ability to create and open temporary tables on a local drive.

Encryption Customizations

By default DBISAM uses the Blowfish block cipher encryption algorithm with 128-bit MD5 hash keys for encryption. However, you may replace the encryption in DBISAM with another 8-byte block cipher algorithm by defining event handlers for the TDBISAMEngine OnCryptoInit, OnEncryptBlock, OnDecryptBlock, and OnCryptoReset events. The OnCryptoInit event is triggered whenever DBISAM needs to initialize the internal block cipher tables using a new key. The OnEncryptBlock event is triggered whenever DBISAM needs to encrypt a block of data, and the OnDecryptBlock event is triggered whenever DBISAM needs to decrypt a block of data. A block of data will always be 8-bytes in length. Finally, the OnCryptoReset event is triggered after every encryption or decryption of a buffer (data record, index page, or BLOB block) in order to reset the cipher data so that it is ready for encrypting or decrypting a new buffer.

Please see the Encryption topic for more information.

Compression Customizations

By default DBISAM uses the ZLIB compression algorithm for compression. However, you may replace the compression in DBISAM with another compression algorithm by defining event handlers for the TDBISAMEngine OnCompress and OnDecompress events. The OnCompress event is triggered whenever DBISAM needs to compress a buffer. The OnDecompress event is triggered whenever DBISAM needs to decompress a buffer.

Please see the Compression topic for more information.

Full Text Indexing Customizations

The full text indexing functionality in DBISAM allows the developer to index the words in string or memo fields for very fast word-based searches. You can define event handlers for the TDBISAMEngine OnTextIndexFilter and OnTextIndexTokenFilter events that allow you to filter the string and memo field

data prior to being indexed by DBISAM. The OnTextIndexFilter event is triggered before DBISAM parses any string or memo fields that are included in the full text index for the table into words using the stop words, space characters, and include characters defined for the table. This allows you to filter the raw data, such as stripping out control codes from HTML, RTF, or other types of document formats. On the other hand, the OnTextIndexTokenFilter event is triggered after any string and memo fields are parsed into words using the stop words, space characters, and include characters defined for the table. This allows you to further filter out certain words based upon conditional rules or custom dictionaries that aren't easily expressed using just the static stop words for the table. Please see the Full Text Indexing topic for more information.

Note
If you add or modify the OnTextIndexFilter or OnTextIndexTokenFilter event handlers when you have existing tables with full text indexing defined for one or more fields, you must be sure to alter the structure of these tables and turn off the full text indexing for all fields. After you have done this, you can then alter the structure of these tables again to turn back on the full text indexing for the desired fields. Doing this will ensure that any existing text is properly handled with the new event handlers and will eliminate the possibility of confusing results when searching on the fields that are part of the full text index. Please see the Creating and Altering Tables topic for more information.

Reserved Customizations

There are certain customizations in the engine that are only for use in fine-tuning specific issues that you may be having with an application and should not be modified unless instructed to do so by Elevate Software. The TDBISAMEngine TableReadLockTimeout, TableWriteLockTimeout, TableTransLockTimeout, TableFilterIndexThreshhold properties should only be modified when instructed to by Elevate Software.

Server-Only Customizations

The following customizations are only available when the TDBISAMEngine EngineType property is set to etServer and the engine is behaving as a database server.

Licensed Connections

You can specify that a maximum number of licensed connections be used for the database server by modifying the TDBISAMEngine ServerLicensedConnections property. The default is 65,535 connections. Setting this property to a lower figure will allow no more than the specified number of connections to be configured as the maximum number of connections for the database server in addition to actually preventing any more than the specified number of connections active on the database server at the same time.

Notification Events

You can define event handlers for the following TDBISAMEngine events to respond to various server conditions:

Event	Description

OnServerStart	This event will be triggered whenever the server starts listening for incoming normal data connections. The server is started via the TDBISAMEngine StartMainServer method or remotely via the TDBISAMSession StartRemoteServer method.
OnServerStop	This event will be triggered whenever the server stops listening for incoming noraml data connections. The server is stopped via the TDBISAMEngine StopMainServer method or remotely via the TDBISAMSession StopRemoteServer method.
OnServerConnect	This event will be triggered whenever a normal data connection is established.
OnServerReconnect	This event will be triggered whenever a normal data connection is re-established by an automatic reconnection by the remote session.
OnServerLogin	This event will be triggered whenever a user logs in on a normal data connection.
OnServerLogout	This event will be triggered whenever a user logs out on a normal data connection.
OnServerDisconnect	This event will be triggered whenever a normal data connection is closed.

Logging Events

DBISAM abstracts all server logging functionality so that you may choose to log server events in any manner that you wish. The default server project that ships with DBISAM uses these events to store the log records in a binary file. You can define event handlers for the following TDBISAMEngine events to customize the logging functionality:

Event	Description
OnServerLogEvent	This event is triggered whenever the server needs to log an event. The log record that is passed to the event handler is defined as a TLogRecord type.
OnServerLogCount	This event is triggered whenever the server needs to get a count of the number of log records in the current log. This event is triggered whenever the TDBISAMEngine GetServerLogCount method is called or the TDBISAMSession GetRemoteLogCount method is called by a remote session.
OnServerLogRecord	This event is triggered whenever the server needs to get a specific log record from the current log. This event is triggered whenever the TDBISAMEngine GetServerLogRecord method is called or the TDBISAMSession GetRemoteLogRecord method is called by a remote session.

Scheduled Events

DBISAM allows the definition of scheduled events for a database server. Scheduled events are stored in the configuration file for the server and are implemented via the TDBISAMEngine OnServerScheduledEvent event. Scheduled events will simply do nothing unless they are actually implemented in the database server via an event handler assigned to this event. Scheduled events are executed in a separate thread in the server, one thread for each currently-executing scheduled event. If you have three scheduled events

that are scheduled for the same time, then the server will create three threads, one for each scheduled event. Any database access within the thread must be done according to the rules for using DBISAM in a multi-threaded application. Please see the Multi-Threaded Applications topic for more information. Also, scheduled events are run as if they are using a local engine accessing databases and tables directly and cannot directly use database names that are defined in the database server configuration. You must use the methods available in the TDBISAMEngine component for retrieving database information for databases for retrieving the information necessary to access server databases and tables in the scheduled event (see the example below).

The following is an example of a scheduled event called "DailyBackup" that calls the TDBISAMDatabase Backup method to backup a databse every day at a certain time:

```
void __fastcall TMyForm::ServerScheduledEvent(TObject *Sender,
      const AnsiString EventName, bool &Completed)
{
   AnsiString TempDescription="";
   AnsiString TempPath="";
   if (AnsiCompareText(EventName,"DailyBackup")==0)
      {
      // Create a new session component, remembering
      // the multi-threading requirements of DBISAM
      // for session names
      TDBISAMSession *TempSession=new TDBISAMSession(this);
      try
         {
         TempSession->SessionName="DailyBackup"+
             IntToStr(GetCurrentThreadID);
         TempSession->Active=true;
         // Create a new database component
         TDBISAMDatabase *TempDatabase=new TDBISAMDatabase(this);
         try
            {
            TempDatabase->SessionName=TempSession->SessionName;
            TempDatabase->DatabaseName="DailyBackup";
            // Get the actual local path for the Main
            // database
            Engine()->GetServerDatabase("Main",
                                        TempDescription,
                                        TempPath);
            TempDatabase->Directory=TempPath;
            TempDatabase->Connected=true;
            TStringList *BackupFiles=new TStringList;
            try
               {
               TempSession->GetTableNames(DatabaseName,BackupFiles);
               Completed=TempDatabase->Backup(
                    IncludeTrailingBackslash(TempPath)+"backup"+
                    StringReplace(DateToStr(Date),"/",
                                  "",[rfReplaceAll])+".bkp",
                    "Daily Backup for "+DateToStr(Date),6,BackupFiles);
               }
            __finally
               {
               delete BackupFiles;
               }
            TempDatabase->Connected=false;
            }
          finally
```

```
            {
            delete TempDatabase;
            }
         }
      __finally
         {
         delete TempSession;
         }
      }
   else
      {
      Completed=true;
      }
}
```

Note
If a scheduled event is not marked as completed by this event handler, it will continue to be executed every minute by the database server until the time range for which it was scheduled is up. For example, if the above scheduled event was scheduled to run every day between 11:00pm and 11:30pm, the database server will attempt to execute the scheduled event until it is either completed or the time exceeds 11:30pm. Also, if an error occurs during the scheduled event execution, the database server will consider the scheduled event not completed. Any time the database server encounters an error in the scheduled event or detects that the scheduled event did not complete it will log this information in the current log.

Server Procedures

DBISAM allows the definition of server-side procedures for a database server. Server-side procedures are stored in the configuration file for the server and are implemented via the TDBISAMEngine OnServerProcedure event. Server-side procedures will simply do nothing unless they are actually implemented in the database server via an event handler assigned to this event. Server-side procedures are executed in the context of the session thread currently running for the remote session that is calling the server-side procedure. Any database access within the server-side procedure must be done according to the rules for using DBISAM in a multi-threaded application. Please see the Multi-Threaded Applications topic for more information. However, unlike scheduled events (see above), server-side procedures are passed a TDBISAMSession component for use in the procedure for retrieving parameters passed in from the remote session and for populating the result parameters that are passed back to the remote session after the procedure is done, as well as sending progress information back to the calling session. This TDBISAMSession component is automatically created and assigned a unique SessionName property to ensure that it can be safely be used in a multi-threaded manner. This session name consists of the user name plus an underscore (_) plus the session ID. Also, server-side procedures are run as if they are using a local engine accessing databases and tables directly and cannot directly use database names that are defined in the database server configuration. You must use the methods available in the TDBISAMEngine component for retrieving database information for databases for retrieving the information necessary to access server databases and tables in the server-side procedure.

The TDBISAMSession RemoteParams property is used both to pass the parameters to the server-side procedure and to return any results to the remote session that called the server-side procedure. The RemoteParams property is a TDBISAMParams object. Be sure to always clear the parameters using the RemoteParams' Clear method before leaving the server-side procedure. Otherwise, the same parameters that were passed to the server-side procedure will be returned to the remote session as results. You can add new results to the RemoteParams property for return to the remote session using the RemoteParams' CreateParam method.

The following is an example of a server-side procedure called "TextFile" that sends a text file back to the remote session that requested it:

```
void __fastcall TMyForm::ServerProcedure(TObject *Sender,
      TDBISAMSession *ServerSession, const AnsiString ProcedureName)
{
   AnsiString TempFileName="";
   if (AnsiCompareText(ProcedureName,"TextFile")==0)
      {
      TempFileName=ServerSession->RemoteParams->
                     ParamByName("FileName")->AsString;
      // Now clear the parameters for use in populating
      // the result parameters
      ServerSession->RemoteParams->Clear;
      if (FileExists(TempFileName))
         {
         // If the file exists, use the TDBISAMParam
         // LoadFromFile method to load the file
         // data into the parameter
         ServerSession->RemoteParams->
            CreateParam(ftMemo,"FileContents")->
            LoadFromFile(TempFileName,ftMemo);
         }
      else
         {
         // If the file doesn't exist, just create a NULL
         // parameter with the correct result name
         ServerSession->RemoteParams->
             CreateParam(ftMemo,"FileContents");
         }
      }
}
```

The ServerSession CurrentServerUser property can be referenced from within a trigger that is being executed when the TDBISAMEngine EngineType property is set to etServer in order to retrieve the current user name.

Note
If a server-side procedure raises any type of exception at all, the database server will send the exception back to the remote session that called it as if the exception occurred in the remote session.

To report progress information back to the calling session during the server-side procedure, use the SendProcedureProgress method of the TDBISAMSession component passed as a parameter to the OnServerProcedure event handler.

2.7 Starting Sessions

Introduction

As already discussed in the DBISAM Architecture topic, the TDBISAMSession component represents a session in DBISAM. The following information will show how to start a session in an application.

Preparing a Local Session for Startup

If a TDBISAMSession component has its SessionType property set to stLocal, then it is considered a local session as opposed to a remote session. There is nothing extra that must be done to prepare a local session for startup.

Preparing a Remote Session for Startup

If a TDBISAMSession component has its SessionType property set to stRemote, then it is considered a remote session as opposed to a local session. Starting a remote session will cause DBISAM to attempt a connection to the database server specified by the RemoteAddress or RemoteHost and RemotePort or RemoteService properties. In addition, the RemoteEncryption property indicates whether the session's connection to the database server will be encrypted using the RemoteEncryptionPassword property. You must set these properties properly before trying to open the remote session or an exception will be raised.

The RemoteAddress and RemoteHost properties are normally mutually exclusive. They can both be specified, but the RemoteHost property will take precedence. The host name used for the server can be specified via the "hosts" text file available from the operating system. In Windows 98, for example, it's located in the Windows directory and is called "hosts.sam". Renaming this file to just "hosts" and adding an entry in it for the database server will allow you to refer to the database server by host name instead of IP address. The following is an example of an entry for a database server running on a LAN:

```
192.168.0.1     DBISAMLANServer
```

This is sometimes more convenient than remembering several IP addresses for different database servers. It also allows the IP address to change without having to modify your application.

The RemotePort and RemoteService properties are also normally mutually exclusive. They can both be specified, but the RemoteService property will take precedence. By default the ports that DBISAM database servers use are:

Port	Usage
12005	Normal access
12006	Administrative access

These ports can be changed, however, so check with your administrator or person in charge of the database server configuration to verify that these are the ports being used.

The service name used for the database server can be specified via the "services" text file available from the operating system. In Windows 98, for example, it's located in the \Windows directory and is called "services". Adding an entry in it for the database server's port will allow you to refer to the server's port by service name instead of port number. The following is an example of an entry for both the normal server

port and the administrative port:

```
DBISAMServer     12005/tcp
DBISAMAdmin     12006/tcp
```

This is sometimes more convenient than remembering the port numbers for different database servers. It also allows the port number to change without having to modify your application.

The RemoteEncryption property can be set to either True or False and determines whether the session's connection to the server will be encrypted or not. If this property is set to True, the RemoteEncryptionPassword property is used to encrypt and decrypt all data transmitted to and from the database server. This property must match the same encryption password that the database server is using or else an exception will be raised when a request is attempted on the server.

Note
When connecting as an administrator to the administrative port of the database server, you must set the RemoteEncryption property to True since administrative connections always require encryption.

If for any reason DBISAM cannot connect to a database server an exception will be raised. The error code that is returned when a connection fails is 11280 and is defined as DBISAM_REMOTECONNECT in the dbisamcn unit (Delphi) or dbisamcn header file (C++). It's also possible for DBISAM to be able to connect to the server, but the connection will be rejected due to the database server's maximum connection setting being reached (11282 and defined as DBISAM_REMOTEMAXCONNECT), the database server not accepting any new logins (11281 and defined as DBISAM_REMOTENOLOGIN), the database server blocking the client workstation's IP address from accessing the server (11283 and defined as DBISAM_REMOTEADDRESSBLOCK), or an encrypted connection being required by the database server (11277 and defined as DBISAM_REMOTEENCRYPTREQ).

The RemoteUser and RemotePassword properties can be used to automate the login to a database server. Every DBISAM database server uses the following default user ID and password if the database server is being started for the first time, or if it is being started with an empty or missing configuration file:

```
User ID: Admin (case-insensitive)
Password: DBAdmin (case-sensitive)
```

Starting a Session

To start a session you must set the TDBISAMSession Active property to True or call its Open method. For a local session (SessionType property is set to stLocal), the session will be opened immediately. As discussed above, for a remote session (SessionType property is set to stRemote), performing this operation will cause the session to attempt a connection to the database server specified by the RemoteAddress or RemoteHost and RemotePort or RemoteService properties. If the RemoteUser and RemotePassword properties are specified and are valid, then neither the OnRemoteLogin event nor the interactive login dialog will be triggered. If these properties are not specified or are not valid, the OnRemoteLogin event will be triggered if there is an event handler assigned to it. If an event handler is not assigned to the OnRemoteLogin event, DBISAM will display an interactive login dialog that will prompt for a user ID and password. All database servers require a user ID and password in order to connect and login. DBISAM will allow for up to 3 login attempts before issuing an exception. The error code that is

returned when a connection fails due invalid login attempts is 11287 and is defined as DBISAM_REMOTEINVLOGIN in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

Note
Any version of DBISAM for Delphi 6 or higher (including C++Builder 6 and higher) requires that you include the DBLogDlg unit to your uses clause in order to enable the display of a default remote login dialog. This is done to allow for DBISAM to be included in applications without linking in the forms support, which can add a lot of unnecessary overhead and also cause unwanted references to user interface libraries. This is not required for Delphi 5 or C++Builder 5, but these versions always include forms support.

The OnStartup event is useful for handling the setting of any pertinent properties for the session before the session is started. This event is called right before the session is started, so it is useful for situations where you need to change the session properties from values that were used at design-time to values that are valid for the environment in which the application is now running. The following is an example of using an OnStartup event handler to set the remote connection properties for a session:

```
void __fastcall TMyForm::MySessionStartup(TObject *Sender)
{
   TRegistry *Registry = new TRegistry;
   try
      {
      Registry->RootKey=HKEY_LOCAL_MACHINE;
      if (Registry->OpenKey("SOFTWARE/My Application",false)
         {
         if (Registry->ReadBool("IsRemote"))
            {
            MySession->SessionType=stRemote;
            MySession->RemoteAddress=Registry->ReadString("RemoteAddress");
            MySession->RemotePort=Registry->ReadString("RemotePort");
            }
         else
            {
            MySession->SessionType=stLocal;
            }
         }
      else
         {
         ShowMessage("Error reading connection information "+
                     "from the registry");
         }
      }
   __finally
      {
      delete Registry;
      }
}
```

Note
You should not call the session's Open method or toggle the Active property from within this event handler. Doing so can cause infinite recursion.

The OnShutdown event can be used for taking specific actions after a session has been stopped. As is the case with the OnStartup event, the above warning regarding the Open method or Active property also applies for the OnShutDown event.

More Session Properties

After a session is started, it can also be used to control certain global settings for all TDBISAMDatabase, TDBISAMQuery, and TDBISAMTable components that link to the session via their SessionName properties. The properties that represent these global settings are detailed below:

Property	Description
ForceBufferFlush	Controls whether the session will automatically force the operating system to flush data to disk after every write operation completed by DBISAM. Please see the Buffering and Caching topic for more information.
LockProtocol	Controls whether the session will use a pessimistic or optimistic locking model when editing records via navigational or SQL methods. Please see the Locking and Concurrency topic for more information.
LockRetryCount	Controls the number of times that the engine will retry a record or table lock before raising an exception. This property is used in conjunction with the LockWaitTime property.
LockWaitTime	Controls the amount of time, in milliseconds, that the engine will wait in-between lock attempts. This property is used in conjuction with the LockRetryCount property.
KeepConnections	Controls whether temporary TDBISAMDatabase components are kept connected even after they are no longer needed. This property has no effect upon a local session, but can result in tremendous performance improvements for a remote session, therefore it defaults to True and should be left as such in most cases.
PrivateDir	Controls where temporary files generated by DBISAM are stored for a local session. This property is ignored for remote sessions.
ProgressSteps	Controls the maximum number of progress events that any batch operation will generate. Setting this property to 0 will cause the suppression of all progress messages.
StrictChangeDetection	Controls whether DBISAM will use strict or lazy change detection for the session. The default is False, or lazy change detection. Please see the Change Detection topic for more information.

Note
You can modify all of the above session properties both before and after a session is started. However, they do not have any effect upon a session until the session is actually started.

2.8 Calling Server-Side Procedures

Introduction

DBISAM allows a database server to be customized via server-side procedures. Remote sessions may then call these server-side procedures in order to isolate batch processes and other types of processing on the database server. This helps reduce network traffic and allow for all-or-nothing processes that will complete regardless of whether the client workstation loses its connection to the database server or goes down unexpectedly. To see how to define the actual server-side procedure on the server, please see the Customizing the Engine topic.

Calling the Procedure

To successfully call a server-side procedure you must be logged into the database server as a user that has been granted rights to execute the server-side procedure that you wish to call. Please see the Server Administration topic for more information.

Before calling the server-side procedure, you must populate the TDBISAMSession RemoteParams property as needed for any parameters to the procedure using the TDBISAMParams CreateParam method, call the TDBISAMSession CallRemoteProcedure method with the proper procedure name (case-insensitive), and then examine any needed return parameters using the RemoteParams property or the TDBISAMSession RemoteParamByName method. The following example shows how you would call a server-side procedure named "Test_Procedure" that accepts an integer and a string:

```
{
   MyRemoteSession->RemoteParams->
       CreateParam(ftInteger,"ID")->AsInteger=10;
   MyRemoteSession->RemoteParams->
       CreateParam(ftString,"Name")->AsInteger="Test";
   try
      {
      // Now call the procedure
      MyRemoteSession->CallRemoteProcedure("Test_Procedure");
      if (MyRemoteSession->RemoteParams->ParamByName("Result")->AsBoolean)
         {
         ShowMessage("The record was added successfully");
         }
      else
         {
         ShowMessage("The record was not added successfully");
         }
      }
   catch
      {
      ShowMessage("There was an error calling the "+
                  "server-side procedure");
      }
}
```

Handling Exceptions in Procedures

If a server-side procedure raises any type of exception at all, the database server will send the exception

back to the remote session and raise it in the context of the CallRemoteProcedure method call. Defining a try..except block (Delphi) or a try..catch block (C++) is the best way to handle these exceptions since you can then respond to them accordingly based upon the server-side procedure that you are calling.

2.9 Opening Databases

Introduction

As already discussed in the DBISAM Architecture topic, the TDBISAMDatabase component represents a database in DBISAM. The following information will show how to open a database in an application.

Preparing a Database for Opening

Before you can open a database using the TDBISAMDatabase component, you must set a couple of properties. The TDBISAMDatabase DatabaseName property is the name given to the database within the application and is required for naming purposes only. For a local database the Directory property should contain a directory name, either in UNC format or using logical drive mapping notation. For a remote database, the RemoteDatabase property will contain the name of a logical database set up on the database server that you are connecting to.

Note
Setting the Directory property for a local database so that it points to an invalid directory and then opening the database will not cause an error. However, an exception will be raised if a TDBISAMTable or TDBISAMQuery component that is linked to the TDBISAMDatabase via its DatabaseName property tries to open a table. The error code that is returned when a table open fails due to the directory or table files not being present is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

Opening a Database

To open a database you must set the TDBISAMDatabase Connected property to True or call its Open method. For a local TDBISAMDatabase component whose SessionName property is linked to a local TDBISAMSession component, the database will cause the local TDBISAMSession to be opened if it is not already, and then the database will be opened. For a remote database whose SessionName property is linked to a remote TDBISAMSession component, performing this operation will cause the remote session to attempt a connection to the database server if it is not already connected. If the connection is successful, the database will then be opened.

The BeforeConnect event is useful for handling the setting of any pertinent properties for the TDBISAMDatabase component before it is opened. This event is triggered right before the database is opened, so it's useful for situations where you need to change the database information from that which was used at design-time to something that is valid for the environment in which the application is now running. The following is an example of a BeforeConnect event handler that is used to set the properties for a TDBISAMDatabase component before it is opened:

```
void __fastcall TMyForm::MyDatabaseBeforeConnect(TObject *Sender)
{
   TRegistry *Registry=new TRegistry(this);
   try
      {
      // Make sure that the DatabaseName is set
      MyDatabase->DatabaseName="MyDatabase";
      // Now set the Directory property to the value
      // from the registry
```

```
      Registry->RootKey=HKEY_LOCAL_MACHINE;
      if (Registry->OpenKey("SOFTWARE/My Application",false)
         {
         MyDatabase->Directory=Registry->ReadString("Directory");
         }
      else
         {
         ShowMessage("Error reading database information "+
                     "from registry");
         }
      }
   __finally
      {
      delete Registry;
      }
}
```

Note
You should not call the TDBISAMDatabase Open method or modify the Connected property from within this event handler. Doing so can cause infinite recursion.

More Database Properties

A TDBISAMDatabase component has one other property of importance that is detailed below:

Property	Description
KeepConnection	Controls whether the database connection is kept active even after it is no longer needed. This property has no effect upon a local session, but can result in tremendous performance improvements for a remote session, therefore it defaults to True and should be left as such in most cases.
KeepTablesOpen	Controls whether the physical tables opened with the database connection are kept open even after they are closed by the application. Setting this property to True can dramatically improve the performance of large SQL scripts and any other operations that involve constantly opening and closing the same tables over and over.

2.10 Transactions

Introduction

DBISAM allows for transactions in order to provide the ability to execute multi-table updates and have them treated as an atomic unit of work. Transactions are implemented logically in the same fashion as most other database engines, however at the physical level there are some important considerations to take into account and these will be discussed here.

Executing a Transaction

A transaction is executed entirely by using the StartTransaction, Commit, and Rollback methods of the TDBISAMDatabase component. A typical transaction block of code looks like this:

```
{
   MyDatabase->StartTransaction();
   try
      {
      // Perform some updates to the table(s) in this database
      MyDatabase->Commit();
      }
   catch
      {
      MyDatabase->Rollback();
      throw;
      }
}
```

Note
It is very important that you always ensure that the transaction is rolled back if there is an exception of any kind during the transaction. This will ensure that the locks held by the transaction are released and other sessions can continue to update data while the exception is dealt with. Also, if you roll back a transaction it is always a good idea to refresh any open TDBISAMTable or TDBISAMQuery components linked to the TDBISAMDatabase component involved in the transaction so that they reflect the current data and not any data from the transaction that was just rolled back. Along with refreshing, you should make sure that any pending inserts or edits for the TDBISAMTable or TDBISAMQuery components are cancelled using the Cancel method before the transaction is rolled back to ensure that the inserts or edits are not accidentally posted using the Post method after the transaction is rolled back (unless that is specifically what you wish to do).

Restricted Transactions

It is also possible with DBISAM to start a restricted transaction. A restricted transaction is one that specifies only certain tables be part of the transaction. The StartTransaction method accepts an optional list of tables that can be used to specify what tables should be involved in the transaction and, subsequently, locked as part of the transaction (see below regarding locking). If this list of tables is nil (the default), then the transaction will encompass the entire database.

The following example shows how to use a restricted transaction on two tables, the Customer and Orders

table:

```
{
   TStringList *TablesList=new TStringList;
   try
      {
      TablesList->Add("Customer");
      TablesList->Add("Orders");
      MyDatabase->StartTransaction(TablesList);
      try
         {
         // Perform some updates to the table(s) in this database
         MyDatabase->Commit();
         }
      catch
         {
         MyDatabase->Rollback();
         throw;
         }
      }
   __finally
      {
      delete TablesList;
      }
}
```

Flushing Data to Disk During a Commit

By default, the Commit method will cause a flush of all data to disk within the operating system, which is eqivalent to calling the FlushBuffers method of all TDBISAMTable or TDBISAMQuery components involved in the transaction that were updated. The Commit method has an optional parameter that controls this called ForceFlush and it defaults to True. Passing False as the ForceFlush parameter will improve the performance of a commit operation at the expense of possible data corruption if the application is improperly terminated after the commit takes place. This is due to the fact that the operating system may wait several minutes before it lazily flushes any modified data to disk. Please see the Buffering and Caching topic for more information.

Locking During a Transaction

When a transaction on the entire database is started, DBISAM acquires a special transaction write lock on the entire database. This prevents any other sessions from adding, updating, or deleting any data from the tables in the database while the current transaction is active. When a restricted transaction is started on a specific set of tables, DBISAM will only acquire this special transaction write lock on the tables specified as part of the transaction. This special transaction write lock is a very important concept since it illustrates the importance of keeping transactions short (not more than a couple of seconds) in DBISAM. However, this special transaction write lock does not prevent other sessions from reading data from the tables involved in the transaction or acquiring record or table locks on the tables involved in the transaction while the current transaction is active. This means that it is still possible for other sessions to cause a TDBISAMTable or TDBISAMQuery Edit or Delete method call within the current transaction to fail due to not being able to acquire the necessary record lock.

Any record locks acquired by calling the TDBISAMTable or TDBISAMQuery Edit or Delete methods during a transaction will remain locked even after a call to the TDBISAMTable or TDBISAMQuery Post method. This is also the case for table locks acquired via the TDBISAMTable LockTable method, which will remain locked

even after a call to the TDBISAMTable UnlockTable method has been made. These locks will be released when the transaction is rolled back or committed, but not until that point.

Opening and Closing Tables

If a transaction on the entire database is active and a new table is opened via the TBISAMTable or TDBISAMQuery components, that table will automatically become part of the active transaction. Unlike a transaction on the entire database, if a table involved in a restricted transaction is not currently open at the time that StartTransaction is called, then an attempt will be made to open it at that time. Also, any tables that are opened during the restricted transaction and not initially specified as part of the restricted transaction will be excluded from the transaction. If a table involved in a transaction, either restricted or not, is closed while the transaction is still active, the table will be kept open internally by DBISAM until the transaction is committed or rolled back, at which point the table will then be closed. However, the TDBISAMTable or TDBISAMQuery component that opened the table originally will indicate that the table is closed.

SQL and Transactions

The INSERT, UPDATE, and DELETE SQL statements implicitly use a restricted transaction on the updated tables if a transaction is not already active. The interval at which the implicit transaction is committed is based upon the record size of the table being updated in the query and the amount of buffer space configured for the TDBISAMEngine component via its MaxTableDataBufferCount and MaxTableDataBufferSize properties. The COMMIT INTERVAL clause can be used within these SQL statements to manually control the interval at which the transaction is committed, and applies both to situations where a transaction was explicitly started by the developer and situations where the transaction was implicitly started by DBISAM. In the case where a transaction was explicitly started by the developer, the absence of a COMMIT INTERVAL clause in the SQL statement being executed will force DBISAM to never commit any of the effects of the SQL statement and leaves this up to the developer to handle after the SQL statement completes. The COMMIT INTERVAL clause can also contain the FLUSH keyword, which indicates that any transaction commit that takes place during the execution of the SQL statement should also force an operating system flush to disk. By default, commits that occur during the execution of SQL statements do not force an operating system flush to disk.

In addition to implicit transactions with the INSERT, UPDATE, and DELETE SQL statements, DBISAM also allows the use of the START TRANSACTION, COMMIT, and ROLLBACK SQL statements.

Incompatible Operations

The following operations are not compatible with transactions and will cause a transaction to commit if encountered during a transaction.

Backing Up and Restoring Databases
Verifying and Repairing Tables
Creating and Altering Tables
Adding and Deleting Indexes from a Table
Optimizing Tables
Upgrading Tables
Deleting Tables
Renaming Tables
Emptying Tables
Copying Tables

Isolation Level

The default and only isolation level for transactions in DBISAM is serialized. This means that only the session in which the transaction is taking place will be able to see any inserts, updates, or deletes made during the transaction. All other sessions will see the data as it existed before the transaction began. Only after the transaction is committed will other sessions see the new inserts, updates, or deletes.

Data Integrity

A transaction in DBISAM is buffered, which means that all inserts, updates, or deletes that take place during a transaction are cached in memory for the current session and are not physically applied to the tables involved in the transaction until the transaction is committed. If the transaction is rolled back, then the updates are discarded. With a local session this allows for a fair degree of stability in the case of a power failure on the local workstation, however it will not prevent a problem if a power failure happens to occur while the commit operation is taking place. Under such circumstances it's very likely that physical and/or logical corruption of the tables involved in the transaction could take place. The only way corruption can occur with a remote session is if the database server itself is terminated improperly during the middle of a transaction commit. This type of occurrence is much more rare with a server than with a workstation.

2.11 Backing Up and Restoring Databases

Introduction

Backing up and restoring databases is accomplished through the TDBISAMDatabase Backup, BackupInfo, and Restore methods. The properties used by the Backup, BackupInfo, and Restore methods include the Connected,
DatabaseName, Directory, and RemoteDatabase properties. The OnBackupProgress, OnBackupLog, OnRestoreProgress, and OnRestoreLog events can be used to track the progress of and log messages about the backup or restore operation. Backing up a database copies all or some of the tables within the database to a compressed or uncompressed backup file. Restoring a database copies all or some of the tables in a compressed or uncompressed backup file into the database, overwriting any tables with the same names that already exist in the database.

Backing Up a Database

To backup a database you must specify the DatabaseName and Directory or RemoteDatabase properties of the TDBISAMDatabase component, set the Connected property to True, and then call the Backup method. If you are backing up a database from a local session then you will specify the Directory property. If you are backing up a database from a remote session then you will specify the RemoteDatabase property. The TDBISAMDatabase component must be open when this method is called. If the TDBISAMDatabase component is closed an exception will be raised.

Note
When the backup executes, it obtains a read lock for the entire database that prevents any sessions from performing any writes to any of the tables in the database until the backup completes. However, since the execution of this method is quite fast the time during which the tables cannot be changed is usually pretty small. To ensure that the database is available as much as possible for updating, it is recommended that you backup the tables to a file on a hard drive and then copy the file to a CD, DVD, or other slower backup device outside of the scope of the database being locked.

The following example shows how to backup a local database using the Backup method:

```
The local database has the following tables:

Table Name
------------------------------------
Customers
Orders
Items
```

```
{
   TStringList *TablesToBackup=new TStringList;
   try
      {
      MyDatabase->DatabaseName="MyDatabase";
      MyDatabase->Directory="d:\\temp";
      TablesToBackup->Add("Customers");
```

```
      TablesToBackup->Add("Orders");
      TablesToBackup->Add("Items");
      if (MyDatabase->Backup("d:\\temp\\"+
                  StringReplace(DateToStr(Date),
                        "/","",[rfReplaceAll])+".bkp",
                  "Daily Backup for "+DateToStr(Date),6,
                  TablesToBackup)
         {
         ShowMessage("Backup was successful");
         }
      else
         {
         ShowMessage("Backup failed");
         }
      }
   __finally
      {
      delete TablesToBackup;
      }
}
```

Note
Remote databases can only reference backup files that are accessible from the database server on which the database resides. You must specify the path to the backup file in a form that the database server can use to open the file.

Tracking the Backup Progress

To take care of tracking the progress of the backup we have provided the OnBackupProgress and OnBackupLog events within the TDBISAMDatabase component. The OnBackupProgress event will report the progress of the backup operation and the OnBackupLog event will report any log messages regarding the backup operation.

Retrieving Information from a Backup File

To retrieve information from a backup file you must specify the DatabaseName and Directory or RemoteDatabase properties of the TDBISAMDatabase component, set the Connected property to True, and then call the BackupInfo method. If you are retrieving information from a backup file from a local session then you will specify the Directory property. If you are retrieving information from a backup file from a remote session then you will specify the RemoteDatabase property. The TDBISAMDatabase component must be open when this method is called. If the TDBISAMDatabase component is closed an exception will be raised.

Note
Remote databases can only reference backup files that are accessible from the database server on which the database resides. You must specify the path to the backup file in a form that the database server can use to open the file.

Restoring a Database

To restore tables to a database you must specify the DatabaseName and Directory or RemoteDatabase properties of the TDBISAMDatabase component, set the Connected property to True, and then call the Restore method. If you are restoring tables to a database from a local session then you will specify the Directory property. If you are restoring tables to a database from a remote session then you will specify the RemoteDatabase property.

Note
The Restore method overwrites any existing tables with names that are the same as those specified in this parameter. You should be very careful when using this method with an existing database to prevent loss of data.

The TDBISAMDatabase component must be open when this method is called. If the TDBISAMDatabase component is closed an exception will be raised.

Note
When the restore executes, it obtains a write lock for the entire database that prevents any sessions from performing any reads or writes from or to any of the tables in the database until the restore completes. However, since the execution of this method is quite fast the time during which the tables cannot be accessed is usually pretty small.

The following example shows how to restore a table to a local database using the Restore method:

```
The local database has the following tables:

Table Name
-------------------------------------
Customers
Orders
Items
```

```
{
   TStringList *TablesToRestore=new TStringList;
   try
      {
      MyDatabase->DatabaseName="MyDatabase";
      MyDatabase->Directory="d:\\temp";
      TablesToRestore->Add("Customers");
      if (MyDatabase->Restore("d:\\temp\\"+
                  StringReplace(DateToStr(Date),
                        "/","",[rfReplaceAll])+".bkp",
                  TablesToRestore)
         {
         ShowMessage("Restore was successful");
         }
      else
         {
         ShowMessage("Restore failed");
         }
      }
```

```
    __finally
       {
       delete TablesToRestore;
       }
}
```

Note
Remote databases can only reference backup files that are accessible from the database server on which the database resides. You must specify the path to the backup file in a form that the database server can use to open the file.

Tracking the Restore Progress

To take care of tracking the progress of the restore we have provided the OnRestoreProgress and OnRestoreLog events within the TDBISAMDatabase component. The OnRestoreProgress event will report the progress of the restore operation and the OnRestoreLog event will report any log messages regarding the restore operation.

2.12 In-Memory Tables

Introduction

DBISAM provides a complete and seamless in-memory table implementation within the same framework as disk-based tables. There are only a few slight differences that should be taken into account when using in-memory tables, and these are detailed below.

DatabaseName Property

The DatabaseName property in the TDBISAMTable and TDBISAMQuery components should always be set to the special in-memory database name "Memory" in order to create or access any in-memory tables. All in-memory tables reside in this same virtual database that is global to the application process. This means that if you create an in-memory table called "mytable" using the TDBISAMTable CreateTable method and then try to create it again elsewhere within the same application, you will receive an error indicating that the table already exists. Because in-memory tables are global to the process, multiple sessions can access and share the same in-memory tables.

Sharing In-Memory Tables

In-memory tables can be shared just like regular disk-based tables. They are also thread-safe and exhibit the same locking and access behaviors.

Creating In-Memory Tables

Just like disk-based tables, in-memory tables must be created before they can be opened.

Deleting In-Memory Tables

Just like disk-based tables, in-memory tables must be deleted if they are no longer needed. If for any reason an in-memory table is not deleted during the execution of an application, DBISAM will automatically delete it when the application process is terminated.

Local and Remote In-Memory Tables

There are no differences between using in-memory tables with local sessions and using in-memory tables with remote sessions other than the fact that in-memory tables created within a remote session are stored on the database server whereas in-memory tables created within a local session are stored locally in the application's memory space.

2.13 Creating and Altering Tables

Introduction

Creating tables and altering the structure of existing tables is accomplished through the CreateTable and AlterTable methods of the TDBISAMTable component. The properties used by the CreateTable and AlterTable methods include the FieldDefs, IndexDefs, DatabaseName, TableName, and Exists properties.

Basic Steps

There are four basic steps that need to be completed when creating a table or altering the structure of an existing table. They are as follows:

1) Define the field definitions using the FieldDefs property, which is a TDBISAMFieldDefs object.

2) Define the index definitions, if any, using the IndexDefs property, which is a TDBISAMIndexDefs object.

3) Set the database and table information using the DatabaseName and TableName properties.

4) Call the CreateTable method if creating a table or the AlterTable method if altering the structure of an existing table.

Defining the field definitions

The FieldDefs property is used to specify which fields to define for the new or existing table. The FieldDefs property is a list of TDBISAMFieldDef objects, each of which contains information about the fields that make up the table. You may add new TDBISAMFieldDef objects using the Add method. There are two different versions of the Add method. One is for use when creating a table and does not accept a FieldNo parameter as the first parameter, and the other is for use when altering the structure of an existing table and requires that you specify the FieldNo parameter as the first parameter. The reason for this difference is that DBISAM uses field numbers (1-based) to distinguish between existing fields in a table and new fields being added. It also uses field numbers in addition to the index position (0-based) of a field definition in the FieldDefs property to determine if a field has been moved in the structure, but still exists. The use of field numbers also allows for the renaming of existing fields in a table without losing data when altering the structure of an existing table.

> **Note**
> You may use the FieldDefs property's Update method to automatically populate the field definitions for the table from table itself specified by the TDBISAMTable TableName property.

The following summarizes how field numbers and the index position of field definitions are used when creating a table or altering the structure of a table:

Value	Rules

Field Number	A field number is 1-based, meaning that it starts at 1 for the first field definition in a table. A field number is automatically assigned for all field definitions when creating a table so it need not be specified and will be ignored if specified. When altering the structure of an existing table, a field number is required for each field definition. As indicated above, using the FieldDefs property's Update method will automatically populate the correct field numbers from an existing table. If adding a new field, the field number should be set to the next largest field number based upon the existing field numbers in the FieldDefs property. For example, if you have 5 field definitions in the FieldDefs property and wish to add another, the new field definition should be specified with 6 as its field number. **Note** The field definitions represented by the FieldDefs property can have gaps in the field numbers when altering the structure of an existing table. The is because it is possible that a given field definition has been deleted, which means that its field number would not be present anywhere in the field definitions. This type of condition is exactly what indicates to DBISAM that the field should be removed from the table structure.
Index Position	An index position is 0-based, meaning that the first field definition is at index position 0, the second field definition at index position 1, etc. When creating a table or altering the structure of an existing table, the index position represents the desired physical position of the field definition in the table after the table creation or alteration takes place. When altering the structure of an existing table, you can move field definitions around to different index positions and leave their field numbers intact. This will indicate to DBISAM that the field has simply moved its position in the structure of the table. You can also use the Insert method to insert a field definition at a specific index position. Like the Add method, there are two versions of the Insert method, one with a FieldNo parameter for use when altering the structure of an existing table and one without for use when creating a table.

Defining the index definitions

The IndexDefs property is used to specify which indexes to define for the new or existing table. The IndexDefs property is a list of TDBISAMIndexDef objects, each of which contains information about the indexes defined for the table. You may add new TDBISAMIndexDef objects using the Add method. Unlike field definitions, DBISAM uses the index name to distinguish between different index definitions, and their index position in the list of index definitions is irrelevant.

Note
You may use the IndexDefs property's Update method to automatically populate the index definitions for the table from table itself specified by the TDBISAMTable TableName property.

Please see the Index Compression topic for more information on the options for index compression in DBISAM.

Setting the Database and Table Information

The DatabaseName and TableName properties are used to specify the name and location of the table to create or the name of the table whose structure you wish to alter. The DatabaseName property can be set to a value that matches the DatabaseName property of an existing TDBISAMDatabase component, or it may directly specify the path to the new or existing table. The TableName property specifies the name of the new or existing table.

Please see the DBISAM Architecture and Opening Tables topics for more information.

Creating the Table

After defining the field and index definitions and setting the database and table information, you can call the CreateTable method to create the actual table. It is usually good practice to also examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to overwrite an existing table. If you do attempt to overwrite an existing table an EDBISAMEngineError exception will be raised. The error code given when a table create fails due to the table already existing is 13060 and is defined as DBISAM_TABLEEXISTS in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

The CreateTable method can be called without any parameters or you may specify many different parameters that set table-wide information for the table such as its description, locale, etc. The following example shows how to create the local "customer" table using the CreateTable method without any additional parameters:

```
{
   MyTable->DatabaseName="d:\\temp";
   MyTable->TableName="customer";
   MyTable->FieldDefs->Clear();
   MyTable->FieldDefs->Add("CustNo",ftFloat,0,true);
   MyTable->FieldDefs->Add("Company",ftString,30,false);
   MyTable->FieldDefs->Add("Addr1",ftString,30,false);
   MyTable->FieldDefs->Add("Addr2",ftString,30,false);
   MyTable->FieldDefs->Add("City",ftString,15,false);
   MyTable->FieldDefs->Add("State",ftString,20,false);
   MyTable->FieldDefs->Add("Zip",ftString,10,false);
   MyTable->FieldDefs->Add("Country",ftString,20,false);
   MyTable->FieldDefs->Add("Phone",ftString,15,false);
   MyTable->FieldDefs->Add("FAX",ftString,15,false);
   MyTable->FieldDefs->Add("Contact",ftString,20,false);
   MyTable->IndexDefs->Clear();
   MyTable->IndexDefs->Add("","CustNo",TIndexOptions() << ixPrimary);
   MyTable->IndexDefs->Add("ByCompany","Company",
                           TIndexOptions() << ixCaseInsensitive,
                           "",icDuplicateByte);
   if (!MyTable->Exists())
      {
```

```
        MyTable->CreateTable();
        }
}
```

Altering the Structure of the Table

After defining the field and index definitions and setting the database and table information, you can call the AlterTable method to alter the structure of the existing table. It is usually good practice to also examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to alter the structure of a non-existent table. If you do attempt to alter the structure of a non-existent table an EDBISAMEngineError exception will be raised. The error code given when a table open fails due to the table not being present is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++). Also, DBISAM requires exclusive access to the table during the process of altering the table's structure and an EDBISAMEngineError exception will be raised if the table cannot be opened exclusively. The error code given when a table open fails due to access problems is 11013 and is defined as DBISAM_OSEACCES in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

The AlterTable method can be called without any parameters or you may specify many different parameters that set table-wide information for the table such as its description, locale, etc. If you wish to leave all of the table-wide information as it currently exists in the table, then you should pass the following TDBISAMTable properties to the AlterTable method (in this order):

LocaleID
UserMajorVersion
UserMinorVersion
Encrypted
Password
Description
IndexPageSize
BlobBlockSize
LastAutoIncValue
TextIndexFields
TextIndexStopWords
TextIndexSpaceChars
TextIndexIncludeChars

These properties can be read from the exising table without requiring the table to be opened first. However, in order for DBISAM to read the Password property of an encrypted DBISAM table or alter the structure of an encrypted DBISAM table in general, the password for the encrypted table must already be defined for the current session or else it must be provided via an event handler assigned to the TDBISAMSession OnPassword event or by the user via the dialog that will be displayed by DBISAM if an event handler is not assigned to this event for the current session. Please see the Opening Tables topic for more information.

Note
Calling the basic version of the AlterTable method without any parameters is not the same as calling the AlterTable method with the above properties as parameters. Calling the AlterTable method with no parameters instructs DBISAM to use the default parameters for all table-wide information.

The following example shows how to alter the local "customer" table's structure using the AlterTable method without any additional parameters. In this example we want to add a LastSaleAmount (a BCD field) to this table's structure in front of the LastSaleDate field and then add a secondary index on this new

LastSaleAmount field to speed up filtering in SQL queries:

```
Customer Table Structure Before Alteration

Field #    Name              DataType     Size
-----------------------------------------------
1          CustomerID        ftString     10
2          CustomerName      ftString     30
3          ContactName       ftString     30
4          Phone             ftString     10
5          Fax               ftString     10
6          EMail             ftString     30
7          LastSaleDate      ftDate       0

Index Name      Fields In Index      Options
-----------------------------------------------
(none)          CustomerID           ixPrimary
```

```
{
   MyTable->DatabaseName="c:\\temp";
   MyTable->TableName="customer";
   // Always make sure the table is closed first
   MyTable->Active=False;
   // Update the field definitions using the
   // existing field definitions from the table
   MyTable->FieldDefs.Update();
   // Same for the index definitions
   MyTable->IndexDefs.Update();
   // Now insert the new field definition.  Notice
   // the index position of 6 which is 0-based and
   // the field number of 8 which is 1-based and
   // equal to the next available field number since
   // there are currently 7 field definitions for this
   // table
   MyTable->FieldDefs.Insert(6,8,"LastSaleAmount",ftBCD,2,false);
   MyTable->IndexDefs.Add("LastSaleAmount","LastSaleAmount",
                          TIndexOptions());
   // Now alter the table's structure
   AlterTable();
}
```

```
Customer Table Structure After Alteration

Field #    Name              DataType     Size
-----------------------------------------------
1          CustomerID        ftString     10
2          CustomerName      ftString     30
3          ContactName       ftString     30
4          Phone             ftString     10
5          Fax               ftString     10
6          EMail             ftString     30
7          LastSaleAmount    ftBCD        2
8          LastSaleDate      ftDate       0
```

```
Index Name      Fields In Index      Options
-----------------------------------------------
(none)          CustomerID           ixPrimary
LastSaleDate    LastSaleDate         (none)
```

In addition to using the TDBISAMTable CreateTable and AlterTable methods for creating and altering the structure of existing tables, DBISAM also allows the use of the CREATE TABLE and ALTER TABLE SQL statements.

Backup Files

Unless the SuppressBackups parameter to the AlterTable method is set to True (default is False), DBISAM will make backups of a table's physical files before altering the structure of a table, except when the following four conditions exist:

1) The only alteration of the structure that has taken place has been a change in the table description or the user-defined major or minor version numbers.

2) The only alteration of the structure that has taken place has been a change in the name of a field or its description.

3) The only alteration of the structure that has taken place has been a change in the name of an index.

4) Any combination of these three conditions.

In all other cases DBISAM will make a backup of each physical file associated with the table whose structure is being altered. Each physical file will have the same root table name but with a different extension. These extensions are as follows:

Original Extension	Backup Extension
.dat (data)	.dbk
.idx (indexes)	.ibk
.blb (BLOBs)	.bbk

Note
There is one exception - if the alteration of the table structure has only changed one of the primary or secondary indexes or the full text index (by changing the full text indexing parameters), then only the index file will be backed up. This is designed in this fashion to speed up the process of altering a table's structure when the only change has been to the index definitions.

To restore these files in case of a mistake, simply rename them to the proper extension or copy them to the original file names. Also, these backup files will get overwritten without warning for each structure alteration that occurs on the table. If you need the backup files for future use it's best to copy them to a separate directory where they will be safe.

The file extensions described above are the default extensions and can be changed. Please see the DBISAM Architecture and Customizing the Engine topics for more information.

Tracking the Progress of the Table Structure Alteration

To take care of tracking the progress of the table structure alteration, we have provided the TDBISAMTable and TDBISAMQuery OnAlterProgress events.

Dealing with Data Loss in the Table Structure Alteration

To take care of dealing with data loss during the alteration of a table's structure, we have provided the TDBISAMTable and TDBISAMQuery OnDataLost events. The OnDataLost event is used to track when data is lost due to field conversions between incompatible types, field constraint failures, field deletions, or key violations resulting from changes in the primary index definition or unique secondary index definitions.

2.14 Upgrading Tables

Introduction

Upgrading tables is accomplished through the UpgradeTable method of the TDBISAMTable component. The properties used by the UpgradeTable method include the DatabaseName, TableName, and Exists properties. Upgrading a table takes table from a previous DBISAM table format and modifies its internal format so that it is compatible with the table format used by the version of DBISAM in use during the upgrade. DBISAM maintains a version number in all tables that indicates to DBISAM what format the table is in. You can use the TDBISAMTable VersionNum property to see what table format version a table is using.

Upgrading a Table

To upgrade a table, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the UpgradeTable method. The table component must be closed and the Active property must be False. It is usually good practice to also examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to upgrade a non-existent table. If you do attempt to upgrade a non-existent table an EDBISAMEngineError exception will be raised. The error code given when a table upgrade fails due to the table not existing is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++). DBISAM will attempt to open the table exclusively before upgrading the table. If another session has the table open then an EDBISAMEngineError exception will be raised when this method is called. The error code given when upgrading a table fails due to the table being open by another session is 11013 and is defined as DBISAM_OSEACCES in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

The following example shows how to upgrade the "customer" table using the UpgradeTable method:

```
{
   MyTable->DatabaseName="d:\\temp";
   MyTable->TableName="customer";
   if (MyTable->Exists)
      {
      MyTable->UpgradeTable();
      }
}
```

Note
If a table is already in the proper format for the current version of DBISAM, this method will do nothing.

In addition to using the TDBISAMTable UpgradeTable method for upgrading tables, DBISAM also allows the use of the UPGRADE TABLE SQL statement.

Logging Upgrade Messages

During the upgrade process, DBISAM will relay detailed log messages regarding the process start and stop times and any information it deems pertinent. You can trap these log messages for further display or analysis via the TDBISAMTable and TDBISAMQuery OnUpgradeLog events.

Tracking the Upgrade Progress

To take care of tracking the progress of the upgrade we have provided the TDBISAMTable and TDBISAMQuery OnUpgradeProgress events.

Backup Files

DBISAM will make backups of a table's physical files before upgrading the table. Each physical file will have the same root table name but with a different extension. These extensions are as follows:

Original Extension	Backup Extension
.dat (data)	.dup
.idx (indexes)	.iup
.blb (BLOBs)	.bup

To restore these files in case of a mistake, simply rename them to the proper extension or copy them to the original file names. Also, these backup files will get overwritten without warning for each upgrade that occurs on the table. If you need the backup files for future use it's best to copy them to a separate directory where they will be safe.

The file extensions described above are the default extensions and can be changed. Please see the DBISAM Architecture and Customizing the Engine topics for more information.

2.15 Deleting Tables

Introduction

Deleting tables is accomplished through the DeleteTable method of the TDBISAMTable component. The properties used by the DeleteTable method include the DatabaseName, TableName, and Exists properties.

Deleting a Table

To delete a table, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the DeleteTable method. The table component must be closed and the Active property must be False. It is usually good practice to also examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to delete a non-existent table. If you do attempt to delete a non-existent table an EDBISAMEngineError exception will be raised. The error code given when a table delete fails due to the table not existing is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

The following example shows how to delete the "customer" table using the DeleteTable method:

```
{
   MyTable->DatabaseName="d:\\temp";
   MyTable->TableName="customer";
   if (MyTable->Exists)
      {
      MyTable->DeleteTable();
      }
}
```

Note
You should be extremely careful when using this method since deleting a table will remove the table and its contents permanently. Be sure to have a backup of your data before using this method in order to avoid any costly mistakes.

In addition to using the TDBISAMTable DeleteTable method for deleting tables, DBISAM also allows the use of the DROP TABLE SQL statement.

2.16 Renaming Tables

Introduction

Renaming tables is accomplished through the RenameTable method of the TDBISAMTable component. The properties used by the RenameTable method include the DatabaseName, TableName, and Exists properties.

Renaming a Table

To rename a table, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the RenameTable method. The table component must be closed and the Active property must be False. It is usually good practice to also examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to rename a non-existent table. If you do attempt to rename a non-existent table an EDBISAMEngineError exception will be raised. The error code given when a table rename fails due to the table not existing is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

The following example shows how to rename the "customer" table to the "oldcustomer" table using the RenameTable method:

```
{
   MyTable->DatabaseName="d:\\temp";
   MyTable->TableName="customer";
   if (MyTable->Exists)
      {
      MyTable->RenameTable("oldcustomer");
      }
}
```

Note
You should be extremely careful when using this method since renaming a table can break applications and cause them to encounter errors when trying to open up a table that no longer exists under the same name.

In addition to using the TDBISAMTable RenameTable method for renaming tables, DBISAM also allows the use of the RENAME TABLE SQL statement.

2.17 Adding and Deleting Indexes from a Table

Introduction

Adding and Deleting indexes is accomplished through the AddIndex, DeleteIndex, and DeleteAllIndexes methods of the TDBISAMTable component. The properties used by these methods include the DatabaseName, TableName, and Exists properties.

Adding an Index

To add an index, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the AddIndex method. The table can be open or closed when this method is called, however if the table is already open it must have been opened exclusively, meaning that the Exclusive property should be set to True. If the Exclusive property is set to False, an EDBISAMEngineError exception will be raised when this method is called. The error code given when an addition of an index fails due to the table not being opened exclusively is 10253 and is defined as DBISAM_NEEDEXCLACCESS in the dbisamcn unit (Delphi) or dbisamcn header file (C++). If the table is closed when this method is called, then DBISAM will attempt to open the table exclusively before adding the index. If another session has the table open then an EDBISAMEngineError exception will be raised when this method is called. The error code given when an addition of an index fails due to the table being open by another session is 11013 and is defined as DBISAM_OSEACCES in the dbisamcn unit (Delphi) or dbisamcn header file (C++). It is usually good practice to also examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to add an index to a non-existent table. If you do attempt to add an index to a non-existent table an EDBISAMEngineError exception will be raised. The error code given when adding an index to a table fails due to the table not existing is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++). If you attempt to add an index with the name of an existing index an EDBISAMEngineError exception will be raised. The error code given when adding an index to a table that already contains an index with the same name is 10027 and is defined as DBISAM_INDEXEXISTS in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

The following is an example of adding a case-insensitive index on the Company field in the "customer" table:

```
{
   MyTable->DatabaseName="d:\\temp";
   MyTable->TableName="customer";
   if (MyTable->Exists)
      {
      MyTable->AddIndex("ByCompany","Company",
                  TIndexOptions() << ixCaseInsensitive,""
                  icDuplicateByte);
      }
}
```

Please see the Index Compression topic for more information on the options for index compression in DBISAM.

In addition to using the TDBISAMTable AddIndex method for adding indexes to tables, DBISAM also allows the use of the CREATE INDEX SQL statement.

Backup Files

DBISAM will make backups of a table's physical index file before adding an index to a table. The physical index file will have the same root table name but a different extension. This extension is as follows:

Original Extension	Backup Extension
.idx (indexes)	.ibk

To restore this files in case of a mistake, simply rename them to the proper extension or copy them to the original file name. Also, this backup file will get overwritten without warning for each index addition or structure alteration that occurs on the table. If you need the backup file for future use it's best to copy it to a separate directory where it will be safe.

The file extensions described above are the default extensions and can be changed. Please see the DBISAM Architecture and Customizing the Engine topics for more information.

Tracking the Progress of the Index Addition

To take care of tracking the progress of the index addition, we have provided the TDBISAMTable and TDBISAMQuery OnIndexProgress events.

Dealing with Data Loss in the Index Addition

To take care of dealing with data loss during the addition of an index, we have provided the TDBISAMTable and TDBISAMQuery OnDataLost events. The OnDataLost event is used to track when data is lost due to key violations resulting from the addition of a primary index or unique secondary index.

Deleting an Index

To delete an index, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the DeleteIndex method. The DeleteIndex method accepts one parameter, the name of the index to delete. If you are deleting the primary index of the table you should use a blank string ('') as the index name parameter. The same rules for exclusive table access that apply to the AddIndex method also apply to the DeleteIndex method. If you attempt to delete an index that does not exist an EDBISAMEngineError exception will be raised. The error code given when deleting an index that does not exist in the table is 10022 and is defined as DBISAM_INVALIDINDEXNAME in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

The following is an example of deleting an index called ByCompany from the "customer" table:

```
{
   MyTable->DatabaseName="d:\\temp";
   MyTable->TableName="customer";
   if (MyTable->Exists)
      {
      MyTable->DeleteIndex("ByCompany");
      }
}
```

In addition to using the TDBISAMTable DeleteIndex method for deleting indexes from tables, DBISAM also allows the use of the DROP INDEX SQL statement.

Deleting All Indexes from a Table

To delete all indexes from a table, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the DeleteAllIndexes method. The same rules for exclusive table access that apply to the DeleteIndex method also apply to the DeleteAllIndexes method.

The following is an example of deleting all indexes from the "customer" table:

```
{
   MyTable->DatabaseName="d:\\temp";
   MyTable->TableName="customer";
   if (MyTable->Exists)
      {
      MyTable->DeleteAllIndexes();
      }
}
```

2.18 Emptying Tables

Introduction

Emptying tables is accomplished through the EmptyTable method of the TDBISAMTable component. The properties used by the EmptyTable method include the DatabaseName, TableName, and Exists properties. Emptying a table very quickly removes all of its records while keeping the structure, including indexes, intact.

Emptying a Table

To empty a table, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the EmptyTable method. The table can be open or closed when this method is called, however if the table is already open it must have been opened exclusively, meaning that the Exclusive property should be set to True. If the Exclusive property is set to False, an EDBISAMEngineError exception will be raised when this method is called. The error code given when emptying a table fails due to the table not being opened exclusively is 10253 and is defined as DBISAM_NEEDEXCLACCESS in the dbisamcn unit (Delphi) or dbisamcn header file (C++). If the table is closed when this method is called, then DBISAM will attempt to open the table exclusively before emptying the table. If another session has the table open then an EDBISAMEngineError exception will be raised when this method is called. The error code given when emptying a table fails due to the table being open by another session is 11013 and is defined as DBISAM_OSEACCES in the dbisamcn unit (Delphi) or dbisamcn header file (C++). It is usually good practice to also examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to empty a non-existent table. If you do attempt to empty a non-existent table an EDBISAMEngineError exception will be raised. The error code given when emptying a table fails due to the table not existing is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

The following example shows how to empty the "customer" table using the EmptyTable method:

```
{
   MyTable->DatabaseName="d:\\temp";
   MyTable->TableName="customer";
   if (MyTable->Exists)
      {
      MyTable->EmptyTable();
      }
}
```

Note
You should be extremely careful when using this method since emptying a table will remove the contents of the table permanently. Be sure to have a backup of your data before using this method in order to avoid any costly mistakes.

In addition to using the TDBISAMTable EmptyTable method for emptying tables, DBISAM also allows the use of the EMPTY TABLE SQL statement.

2.19 Copying Tables

Introduction

Copying tables is accomplished through the CopyTable method of the TDBISAMTable component. The properties used by the CopyTable method include the DatabaseName, TableName, and Exists properties. By default, copying a table copies the entire structure and specified contents of a table to a new table. The records that are copied can be controlled by setting a range or filter on the source table being copied prior to calling the CopyTable method. You can also specify False for the last CopyData parameter in order to only copy the table structure and not the table contents.

Note
The CopyTable method acquires a read lock on the source table at the beginning of the copy operation and does not release it until the copy operation is complete. This is done to make sure that no other sessions modify the data as well as make sure that the data that is copied is logically consistent with the original table and does not contain partial updates. Please see the Locking and Concurrency topic for more information.

Copying a Table

To copy a table, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the CopyTable method. The table can be open or closed when this method is called, and the table does not need to be opened exclusively (Exclusive property=True). If the table is closed when this method is called, then the DBISAM engine will attempt to open the table before copying it. It is usually good practice to also examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to copy a non-existent table. If you do attempt to copy a non-existent table an EDBISAMEngineError exception will be raised. The error code given when copying a table fails due to the table not existing is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

The following example shows how to copy the "customer" table to the "newcust" table in the same database directory using the CopyTable method:

```
{
   MyTable->DatabaseName="d:\\temp";
   MyTable->TableName="customer";
   if (MyTable->Exists)
      {
      MyTable->CopyTable("d:\\temp","newcust");
      }
}
```

Note
When copying tables in a local session, you must specify the first database name parameter to the CopyTable method as a local database directory. When copying tables in a remote session, you must specify the first database name parameter to the CopyTable method as a database defined on the database server. You cannot copy tables on a database server to local tables or vice-versa. Please see the DBISAM Architecture topic for more information.

The CopyTable operation can also be performed on a table that is already open and has a range or filter set. This is useful for limiting the copied records to a certain criteria. Please see the Setting Ranges on Tables and Setting Filters on Tables and Query Result Sets topics for more information.

Tracking the Copy Progress

To take care of tracking the progress of the copy we have provided the TDBISAMTable OnCopyProgress event.

2.20 Optimizing Tables

Introduction

Optimizing tables is accomplished through the OptimizeTable method of the TDBISAMTable component. The properties used by the OptimizeTable method include the DatabaseName, TableName, and Exists properties. Optimizing a table will physically re-order a table's records based upon a specific index in order to improve read-ahead performance and will also physically remove any empty space from a table. By default the index used for the re-ordering of the table records is the primary index.

Optimizing a Table

To optimize a table, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the OptimizeTable method. The table component must be closed and the Active property must be False. It is usually good practice to also examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to upgrade a non-existent table. If you do attempt to upgrade a non-existent table an EDBISAMEngineError exception will be raised. The error code given when a table upgrade fails due to the table not existing is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++). DBISAM will attempt to open the table exclusively before optimizing the table. If another session has the table open then an EDBISAMEngineError exception will be raised when this method is called. The error code given when optimizing a table fails due to the table being open by another session is 11013 and is defined as DBISAM_OSEACCES in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

The following example shows how to optimize the "customer" table using the OptimizeTable method:

```
{
   MyTable->DatabaseName="d:\\temp";
   MyTable->TableName="customer";
   if (MyTable->Exists)
      {
      MyTable->OptimizeTable();
      }
}
```

In addition to using the TDBISAMTable OptimizeTable method for optimizing tables, DBISAM also allows the use of the OPTIMIZE TABLE SQL statement.

Tracking the Optimize Progress

To take care of tracking the progress of the optimization we have provided the TDBISAMTable and TDBISAMQuery OnOptimizeProgress events.

Backup Files

By default, DBISAM will make backups of a table's physical files before optimizing the table. You can turn this off via the second parameter to the OptimizeTable method. Each physical file will have the same root table name but with a different extension. These extensions are as follows:

Original Extension	Backup Extension

.dat (data)	.dbk
.idx (indexes)	.ibk
.blb (BLOBs)	.bbk

To restore these files in case of a mistake, simply rename them to the proper extension or copy them to the original file names. Also, these backup files will get overwritten without warning for each optimization that occurs on the table. If you need the backup files for future use it's best to copy them to a separate directory where they will be safe.

The file extensions described above are the default extensions and can be changed. Please see the DBISAM Architecture and Customizing the Engine topics for more information.

2.21 Verifying and Repairing Tables

Introduction

Verifying and repairing tables is accomplished through the VerifyTable and RepairTable methods of the TDBISAMTable component. The properties used by the VerifyTable and RepairTable methods include the DatabaseName, TableName, and Exists properties. Verifying a table will check the table for any corruption and indicate whether the table is valid or whether it is corrupted. Repairing a table will perform the actual repair of a table, which is primarily making sure that the table is structurally sound, and indicate whether the table was valid or whether it was corrupted.

Verifying a Table

To verify a table, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the VerifyTable method. The VerifyTable method returns True if the table is valid and False if the table is corrupted. The table component must be closed and the Active property must be False. It is usually good practice to also examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to verify a non-existent table. If you do attempt to verify a non-existent table an EDBISAMEngineError exception will be raised. The error code given when a table verification fails due to the table not existing is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++). DBISAM will attempt to open the table exclusively before verifying the table. If another session has the table open then an EDBISAMEngineError exception will be raised when this method is called. The error code given when verifying a table fails due to the table being open by another session is 11013 and is defined as DBISAM_OSEACCES in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

The following example shows how to verify the "customer" table using the VerifyTable method:

```
{
   MyTable->DatabaseName="d:\\temp";
   MyTable->TableName="customer";
   if (MyTable->Exists)
      {
      if (MyTable->VerifyTable())
         {
         ShowMessage("Table is valid")
         }
      else
         {
         ShowMessage("Table is corrupted");
         }
      }
}
```

In addition to using the TDBISAMTable VerifyTable method for verifying tables, DBISAM also allows the use of the VERIFY TABLE SQL statement.

Logging Verification Messages

During the verification process, DBISAM will relay detailed log messages regarding the process start and stop times and any information it deems pertinent. You can trap these log messages for further display or

analysis via the TDBISAMTable and TDBISAMQuery OnVerifyLog events.

Tracking the Verification Progress

To take care of tracking the progress of the verification we have provided the TDBISAMTable and TDBISAMQuery OnVerifyProgress events.

Repairing a Table

To repair a table, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the RepairTable method. The RepairTable method returns True if the table was valid and False if the table was corrupted and needed to be repaired. The table component must be closed and the Active property must be False. It is usually good practice to also examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to repair a non-existent table. If you do attempt to repair a non-existent table an EDBISAMEngineError exception will be raised. The error code given when a table verification fails due to the table not existing is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++). DBISAM will attempt to open the table exclusively before repairing the table. If another session has the table open then an EDBISAMEngineError exception will be raised when this method is called. The error code given when repairing a table fails due to the table being open by another session is 11013 and is defined as DBISAM_OSEACCES in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

The following example shows how to repair the "customer" table using the RepairTable method:

```
{
   MyTable->DatabaseName="d:\\temp";
   MyTable->TableName="customer";
   if (MyTable->Exists)
      {
      if (MyTable->RepairTable())
         {
         ShowMessage("Table was valid")
         }
      else
         {
         ShowMessage("Table was corrupted, check the log "+
                     "messages for repair status");
         }
      }
}
```

In addition to using the TDBISAMTable RepairTable method for repairing tables, DBISAM also allows the use of the REPAIR TABLE SQL statement.

Logging Repair Messages

During the repair process, DBISAM will relay detailed log messages regarding the process start and stop times and any information it deems pertinent. You can trap these log messages for further display or analysis via the TDBISAMTable and TDBISAMQuery OnRepairLog events.

Tracking the Repair Progress

To take care of tracking the progress of the repair we have provided the TDBISAMTable and

TDBISAMQuery OnRepairProgress events.

2.22 Opening Tables

Introduction

Opening tables can be accomplished through the Open method of the TDBISAMTable component, or by setting the Active property to True. Before opening a table, however, you must first specify the location of the table and the table name itself. The location of the table is specified in the DatabaseName property of the TDBISAMTable component, and the table name is specified in the TableName property.

Setting the DatabaseName Property

You may specify the DatabaseName property using two different methods:

1) The first method is to set the DatabaseName property of the TDBISAMTable component to the DatabaseName property of an existing TDBISAMDatabase component within the application. In this case the database location will come from either the Directory property or the RemoteDatabase property depending upon whether the TDBISAMDatabase has its SessionName property set to a local or remote session. Please see the Starting Sessions and Opening Databases topics for more information. The following example shows how to use the DatabaseName property to point to an existing TDBISAMDatabase component for the database location:

```
{
   MyDatabase->DatabaseName="AccountingDB";
   MyDatabase->Directory="c:\\acctdata";
   MyDatabase->Connected=true;
   MyTable->DatabaseName="AccountingDB";
   MyTable->TableName="ledger";
   MyTable->Active=true;
}
```

Note
The above example does not assign a value to the SessionName property of either the TDBISAMDatabase or TDBISAMTable component because leaving this property blank for both components means that they will use the default session that is automatically created by DBISAM when the engine is initialized. This session is, by default, a local, not remote, session named "Default" or "". Please see the Starting Sessions topic for more information.

Another useful feature is using the BeforeConnect event of the TDBISAMDatabase component to dynamically set the Directory or RemoteDatabase property before the TDBISAMDatabase component attempts to connect to the database. This is especially important when you have the Connected property for the TDBISAMDatabase component set to True at design-time during application development and wish to change the Directory or RemoteDatabase property before the connection is attempted when the application is run.

2) The second method is to enter the name of a local directory, if the TDBISAMTable component's SessionName property is set to a local session, or remote database, if the TDBISAMTable component's SessionName property is set to a remote session, directly into the DatabaseName property. In this case a temporary database component will be automatically created, if needed, for the database specified and automatically destroyed when no longer needed. The following example shows how to use the DatabaseName property to point directly to the desired database location without referring to a

TDBISAMDatabase component:

```
{
   MySession->SessionName="Remote";
   MySession->SessionType=stRemote;
   MySession->RemoteAddress="192.168.0.2";
   MySession->Active=true;
   MyTable->SessionName="Remote";
   MyTable->DatabaseName="AccountingDB";
   MyTable->TableName="ledger";
   MyTable->Active=true;
}
```

Note
The above example uses a remote session called "Remote" to connect to a database server at the IP address "192.168.0.2". Using a remote session in this fashion is not specific to this method. We could have easily used the same technique with the TDBISAMDatabase component and its SessionName and RemoteDatabase properties to connect the database in the first example to a remote session instead of the default local session created by the engine. Also, database names are defined on a database server using the remote administration facilities in DBISAM. Please see the Server Administration topic for more information.

Exclusive and ReadOnly Open Modes

In the above two examples we have left the Exclusive and ReadOnly properties of the TDBISAMTable component at their default value of False. However, you can use these two properties to control how the table is opened and how that open affects the ability of other sessions and users to open the same table.

When the Exclusive property is set to True, the table specified in the TableName property will be opened exclusively when the Open method is called or the Active property is set to True. This means that neither the current session nor any other session or user may open this table again without causing an EDBISAMEngineError exception. It also means that the table open will fail if anyone else has the table opened either shared (Exclusive=False) or exclusively (Exclusive=True). The error code given when a table open fails due to access problems is 11013 and is defined as DBISAM_OSEACCES in the dbisamcn unit (Delphi) or dbisamcn header file (C++). The following example shows how to trap for such an exception using a try..except block (Delphi) or try..catch block (C++) and display an appropriate error message to the user:

```
{
      {
      MySession->SessionName="Remote";
      MySession->SessionType=stRemote;
      MySession->RemoteAddress="192.168.0.2";
      MySession->Active=true;
      MyDatabase->SessionName="Remote";
      MyDatabase->DatabaseName="AccountingData";
      MyDatabase->RemoteDatabase="AccountingDB";
      MyDatabase->Connected=True;
      MyTable->SessionName="Remote";
      // We're using a database component for the database
      // location, so we use the same value as the DatabaseName
```

```
      // property for the TDBISAMDatabase component above, not
      // the same value as the RemoteDatabase property, which
      // is the name of the database as defined on the DBISAM
      // database server
      MyTable->DatabaseName="AccountingData";
      MyTable->TableName="ledger";
      MyTable->Exclusive=true;
      MyTable->ReadOnly=False;
      try
         {
         MyTable->Open();
         }
      catch(const Exception &E)
         {
         if (dynamic_cast<EDatabaseError*>(E) &
             dynamic_cast<EDBISAMEngineError*>(E))
            {
            if (dynamic_cast<EDBISAMEngineError&>(*E)->ErrorCode==
                DBISAM_OSEACCES)
               {
               ShowMessage("Cannot open table "+MyTable->TableName+
                    ", another user has the table open already");
               }
            else
               {
               ShowMessage("Unknown or unexpected "+
                    "database engine error # +IntToStr(
                    dynamic_cast<EDBISAMEngineError&>(*E)->ErrorCode));
               }
            }
         else
            {
            ShowMessage("Unknown or unexpected "+
                    "error has occurred");
            }
         }
      }
}
```

Note
Regardless of whether you are trying to open a table exclusively, you can still receive this exception if another user or application has opened the table exclusively.

When the ReadOnly property is set to True, the table specified in the TableName property will be opened read-only when the Open method is called or the Active property is set to True. This means that the TDBISAMTable component will not be able to modify the contents of the table until the table is closed and re-opened with write access (ReadOnly=False). If any of the physical files that make up the table are marked read-only at the operating system level (such as is the case with CD-ROMs) then DBISAM automatically detects this condition and sets the ReadOnly property to True. DBISAM is also able to do extensive read buffering on any table that is marked read-only at the operating system level, so if your application is only requiring read-only access then it would provide a big performance boost to mark the tables as read-only at the operating system level. Finally, if security permissions for any of the physical files that make up the table prevent DBISAM from opening the table with write access, then DBISAM will also automatically detect this condition and set the ReadOnly property to True.

Table Locale Support

It is possible that a table was created using a specific LocaleID that is not available or installed in the operating system currently in use. In such a case this will cause an EDBISAMEngineError exception to be raised when the table is opened. The error code given when a table open fails due to locale support problems is 15878 and is defined as DBISAM_CANNOTLOADLDDRV in the dbisamcn unit (Delphi) or dbisamcn header file (C++). The table cannot be opened until the locale support is installed or the table locale is altered using an operating system with the proper local support.

Opening In-Memory Tables

Opening in-memory tables is the same as opening disk-based tables except for one slight difference. In-memory tables, regardless of whether they are used in a local or remote session, use a special "Memory" database name instead of a normal database name. This special database is always present and the current session always has all rights to the database.

Note
Because in-memory tables in DBISAM act like regular disk-based tables, you must first create the table using the TDBISAMTable CreateTable method and delete the table using the TDBISAMTable DeleteTable method to get rid of the table. Also, all sharing and locking restrictions also apply to in-memory tables just as they do with disk-based tables. Please see the In-Memory Tables topic for more information.

Opening Encrypted Tables

When a table is marked as encrypted and given a password, its contents are then encrypted using this password and any subsequent attempts to open the table only succeed if this password (in this order):

1) Is present in the in-memory list of passwords for the current TDBISAMSession component. You can use the AddPassword, RemovePassword, and RemoveAllPasswords methods to add and remove passwords for the current session.

2) Is provided on-demand through the OnPassword event of the current session.

3) Is provided on-demand through a visual password dialog that will be displayed if the OnPassword event is not assigned an event handler.

Note
When opening a table inside of a TDBISAMEngine scheduled event (OnServerScheduledEvent event) or server procedure (OnServerProcedure event) a visual password dialog will not be displayed and you must either use the TDBISAMSession AddPassword method for adding passwords before trying to open the table or use the OnPassword event to provide passwords for tables as needed or any attempts to open encrypted tables will fail. Also, any version of DBISAM for Delphi 6 or higher (including C++Builder 6 and higher) requires that you include the DBPWDlg unit to your uses clause in order to enable the display of a default password dialog. This is done to allow for DBISAM to be included in applications without linking in the forms support, which can add a lot of unnecessary overhead and also cause unwanted references to user interface libraries. This is not required for Delphi 5 or C++Builder 5, but these versions always include forms support.

The TDBISAMTable Encrypted property will indicate whether a given table is encrypted with a password.

This property does not require that the table be open before accessing it. DBISAM will automatically attempt to open the table, read the encrypted status, and return the value of this property. The Password property will indicate the password for the table in the same manner provided that the table can be opened automatically with the correct password for the current session, as indicated above.

Please see the Creating and Altering Tables topics for more information on creating encrypted tables.

2.23 Closing Tables

Introduction

Closing tables can be accomplished through the Close method of the TDBISAMTable component, or by setting the Active property to False.

The following example shows how to use the Close method to close a table:

```
{
   MyTable->Close();
}
```

Note
Once a table is closed you cannot perform any operations on the table until the table is opened again. The exception to this would be if you are trying to perform an operation that requires the table to be closed, such as repairing or optimizing a table.

2.24 Executing SQL Queries

Introduction

Executing SQL queries is accomplished through the ExecSQL and Open methods of the TDBISAMQuery component, or by setting the Active property to True. Before executing a query you must first specify the location of the table(s) referenced in the query. The location of the table(s) is specified in the DatabaseName property of the TDBISAMQuery component. The actual SQL for the query is specified in the SQL property. Please see the Overview topic in the SQL Reference for more information. You may select whether you want a live or canned query via the RequestLive property. Please see the Live Queries and Canned Queries topic for more information.

Setting the DatabaseName Property

You may specify the DatabaseName property using two different methods:

1) The first method is to set the DatabaseName property of the TDBISAMQuery component to the DatabaseName property of an existing TDBISAMDatabase component within the application. In this case the database location will come from either the Directory property or the RemoteDatabase property depending upon whether the TDBISAMDatabase has its SessionName property set to a local or remote session. Please see the Starting Sessions and Opening Databases topics for more information. The following example shows how to use the DatabaseName property to point to an existing TDBISAMDatabase component for the database location:

```
{
   MyDatabase->DatabaseName="AccountingDB";
   MyDatabase->Directory="c:\\acctdata";
   MyDatabase->Connected=true;
   MyQuery->DatabaseName="AccountingDB";
   MyQuery->SQL->Clear();
   MyQuery->SQL->Add("SELECT * FROM ledger");
   MyQuery->Active=true;
}
```

Note
The above example does not assign a value to the SessionName property of either the TDBISAMDatabase or TDBISAMQuery component because leaving this property blank for both components means that they will use the default session that is automatically created by DBISAM when the engine is initialized. This session is, by default, a local, not remote, session named "Default" or "". Please see the Starting Sessions topic for more information.

Another useful feature is using the BeforeConnect event of the TDBISAMDatabase component to dynamically set the Directory or RemoteDatabase property before the TDBISAMDatabase component attempts to connect to the database. This is especially important when you have the Connected property for the TDBISAMDatabase component set to True at design-time during application development and wish to change the Directory or RemoteDatabase property before the connection is attempted when the application is run.

2) The second method is to enter the name of a local directory, if the TDBISAMQuery component's

SessionName property is set to a local session, or remote database, if the TDBISAMQuery component's SessionName property is set to a remote session, directly into the DatabaseName property. In this case a temporary database component will be automatically created, if needed, for the database specified and automatically destroyed when no longer needed. The following example shows how to use the DatabaseName property to point directly to the desired database location without referring to a TDBISAMDatabase component:

```
{
   MySession->SessionName="Remote";
   MySession->SessionType=stRemote;
   MySession->RemoteAddress="192.168.0.2";
   MySession->Active=true;
   MyQuery->SessionName="Remote";
   MyQuery->DatabaseName="AccountingDB";
   MyQuery->SQL->Clear();
   MyQuery->SQL->Add("SELECT * FROM ledger");
   MyQuery->Active=true;
}
```

Note
The above example uses a remote session called "Remote" to connect to a database server at the IP address "192.168.0.2". Using a remote session in this fashion is not specific to this method. We could have easily used the same technique with the TDBISAMDatabase component and its SessionName and RemoteDatabase properties to connect the database in the first example to a remote session instead of the default local session created by the engine. Also, database names are defined on a database server using the remote administration facilities in DBISAM. Please see the Server Administration topic for more information.

Setting the SQL Property

The SQL statement or statements are specified via the SQL property of the TDBISAMQuery component. The SQL property is a TStrings object. You may enter one SQL statement or multiple SQL statements by using the Add method of the SQL property to specify the SQL statements line-by-line. You can also assign the entire SQL to the Text property of the SQL property. If specifying multiple SQL statements, be sure to separate each SQL statement with a semicolon (;). Multiple SQL statements in one execution is referred to as a script. There is no limit to the number of SQL statements that can be specified in the SQL property aside from memory constraints.

Whenever the SQL property is modified, any event handler assigned to the TDBISAMQuery OnSQLChanged property will be executed.

When dynamically building SQL statements that contain literal string constants, you can use the TDBISAMEngine QuotedSQLStr method to properly format and escape any embedded single quotes or non-printable characters in the string. For example, suppose you have a TMemo component that contains the following string:

```
This is a
test
```

The string contains an embedded carriage-return and line feed, so it cannot be specified directly without causing an error in the SQL statement.

To build an SQL INSERT statement that inserts the above string into a memo field, you should use the following code:

```
MyDBISAMQuery->SQL->Text="INSERT INTO MyTable "+
   "(MyMemoField) VALUES ("+
   Engine()->QuotedSQLStr(MyMemo.Lines.Text)+")";
```

Note
If re-using the same TDBISAMQuery component for multiple query executions, please be sure to call the SQL property's Clear method to clear the SQL from the previous query before calling the Add method to add more SQL statement lines.

Preparing the Query

By default DBISAM will automatically prepare a query before it is executed. However, you may also manually prepare a query using the TDBISAMQuery Prepare method. Once a query has been prepared, the Prepared property will be True. Preparing a query parses the SQL, opens all referenced tables, and prepares all internal structures for the execution of the query. You should only need to manually prepare a query when executing a parameterized query. Please see the Parameterized Queries topic for more information.

Executing the Query

To execute the query you should call the TDBISAMQuery ExecSQL or Open methods, or you should set the Active property to True. Setting the Active property to True is the same as calling the Open method. The difference between using the ExecSQL and Open methods is as follows:

Method	Usage
ExecSQL	Use this method when the SQL statement or statements specified in the SQL property may or may not return a result set. The ExecSQL method can handle both situations.
Open	Use this method only when you know that the SQL statement or statements specified in the SQL property will return a result set. Using the Open method with an SQL statement that does not return a result set will result in an EDatabaseError exception being raised with an error message "Error creating table handle".

Note
The SQL SELECT statement is the only statement that returns a result set. All other types of SQL statements do not.

The following example shows how to use the ExecSQL method to execute an UPDATE SQL statement:

```
{
   MyDatabase->DatabaseName="AccountingDB";
   MyDatabase->Directory="c:\\acctdata";
   MyDatabase->Connected=true;
   MyQuery->DatabaseName="AccountingDB";
   MyQuery->SQL->Clear();
   MyQuery->SQL->Add("UPDATE ledger SET AccountNo=100");
   MyQuery->SQL->Add("WHERE AccountNo=300");
   MyQuery->ExecSQL();
}
```

Retrieving Query Information

You can retrieve information about a query both after the query has been prepared and after the query has been executed. The following properties can be interrogated after a query has been prepared or executed:

Property	Description
SQLStatementType	Indicates the type of SQL statement currently ready for execution. If the TDBISAMQuery SQL property contains multiple SQL statements (a script), then this property represents the type of the current SQL statement about to be executed. You can assign an event handler to the TDBISAMQuery BeforeExecute event to interrogate the SQLStatementType property before each SQL statement is executed in the script.
TableName	Indicates the target table of the SQL statement currently ready for execution. If the TDBISAMQuery SQL property contains multiple SQL statements (a script), then this property represents the target table of the current SQL statement about to be executed. You can assign an event handler to the TDBISAMQuery BeforeExecute event to interrogate the TableName property before each SQL statement is executed in the script.

The following properties can only be interrogated after a query has been executed:

Property	Description

Plan	Contains information about how the current query was executed, including any optimizations performed by DBISAM. This information is very useful in determining how to optimize a query further or to simply figure out what DBISAM is doing behind the scenes. If there is more than one SQL statement specified in the TDBISAMQuery SQL property (a script) then this property indicates the query plan for the last SQL statement executed. You can assign an event handler to the TDBISAMQuery AfterExecute event to interrogate the Plan property after each SQL statement is executed in the script. The Plan property is cleared before each new SQL statement is executed. **Note** Query plans are only generated for SQL SELECT, INSERT, UPDATE, or DELETE statements.
RowsAffected	Indicates the number of rows affected by the current query. If there is more than one SQL statement specified in the TDBISAMQuery SQL property (a script) then this property indicates the cumulative number of rows affected for all SQL statements executed so far. You can assign an event handler to the TDBISAMQuery BeforeExecute and/or AfterExecute events to interrogate the RowsAffected property before and/or after each SQL statement is executed in the script.
ExecutionTime	Indicates the amount of execution time in seconds consumed by the current query. If there is more than one SQL statement specified in the TDBISAMQuery SQL property (a script) then this property indicates the cumulative execution time for all SQL statements executed so far. You can assign an event handler to the TDBISAMQuery BeforeExecute and/or AfterExecute events to interrogate the ExecutionTime property before and/or after each SQL statement is executed in the script.

The following example shows how to use the ExecSQL method to execute an UPDATE SQL statement and report the number of rows affected as well as how long it took to execute the statement:

```
{
   MyDatabase->DatabaseName="AccountingDB";
   MyDatabase->Directory="c:\\acctdata";
   MyDatabase->Connected=true;
   MyQuery->DatabaseName="AccountingDB";
   MyQuery->SQL->Clear();
   MyQuery->SQL->Add("UPDATE ledger SET AccountNo=100");
   MyQuery->SQL->Add("WHERE AccountNo=300");
   MyQuery->ExecSQL();
   ShowMessage(IntToStr(MyQuery->RowsAffected)+
               " rows updated in "+
               FloatToStr(MyQuery->ExecutionTime)+" seconds");
}
```

Trapping for Errors

To take care of trapping for errors during the preparation or execution of queries we have provided the OnQueryError event. Whenever an exception is encountered by DBISAM during the preparation or execution of a query, the exception is passed to the event handler assigned to this event. If there is no event handler assigned to this event, DBISAM will go ahead and raise the exception. You may set the Action parameter of this event to aaAbort in your event handler to indicate to DBISAM that you want to abort the preparation or execution of the entire query, not just the current SQL statement being prepared or executed. You may set the Action parameter of this event to aaContinue to indicate to DBISAM that you want to skip the current SQL statement and continue on with the next SQL statement, if present. This is especially useful for scripts because it gives you the ability to continue on with a script even though one or more of the SQL statements in the script may have encountered an error. Finally, you may set the Action parameter of this event to aaRetry to indicate to DBISAM that you want to retry the current SQL statement. This is especially useful in situations where the application encounters a record lock error during an SQL UPDATE or DELETE statement.

Note
If you use the START TRANSACTION statement within an SQL script, and the script encounters an error in one of the subsequent SQL statements before reaching a COMMIT or ROLLBACK statement, the transaction will be implicitly rolled back if:

1) An OnQueryError event handler is not assigned to the TDBISAMQuery component being used

OR

2) The OnQueryError event handler sets the Action parameter to aaAbort, indicating that the script should immediately terminate.

Tracking the Progress of a Query

To take care of tracking the progress of a query execution we have provided the TDBISAMQuery OnQueryrogress event. You may set the Abort parameter of this event to True in your event handler to indicate to DBISAM that you wish to abort the execution of the current SQL statement.

Note
The percentage of progress reported via the OnQueryProgress event is restarted for every SQL statement specified in the TDBISAMQuery SQL property, so setting the Abort parameter to True will only abort the current SQL statement and not the entire script. Also, the OnQueryProgress event will not be triggered for a live query result. Please see the Live Queries and Canned Queries topic for more information.

SQL-Specific Events

There are certain events that will be triggered when specific SQL statements are executed using the TDBISAMQuery component. These are as follows:

SQL Statement	Events

SELECT	OnQueryProgress
INSERT	None
UPDATE	None
DELETE	None
CREATE TABLE	None
ALTER TABLE	OnAlterProgress OnDataLost
EMPTY TABLE	None
OPTIMIZE TABLE	OnOptimizeProgress
EXPORT TABLE	OnExportProgress
IMPORT TABLE	OnImportProgress
VERIFY TABLE	OnVerifyProgress OnVerifyLog
REPAIR TABLE	OnRepairProgress OnRepairLog
UPGRADE TABLE	OnUpgradeProgress OnUpgradeLog
DROP TABLE	None
RENAME TABLE	None
CREATE INDEX	OnIndexProgress OnDataLost
DROP INDEX	None
START TRANSACTION	None
COMMIT	None
ROLLBACK	None

2.25 Live Queries and Canned Queries

Introduction

DBISAM generates two types of query result sets depending upon the makeup of a given SELECT SQL statement:

Type	Description
Live	The result set is editable and all changes are reflected in the source table.
Canned	The result set is editable but changes are not reflected in the source table(s).

The following rules determine whether a query result set will be live or canned. Please see the Executing SQL Queries for more information on executing queries.

Single-table queries

Queries that retrieve data from a single table will generate a live result set provided that:

1) The TDBISAMQuery RequestLive property is set to True.

2) There is no DISTINCT keyword in the SELECT SQL statement.

3) Everything in the SELECT clause is a simple column reference or a calculated column, and no aggregation or calculated BLOB columns are allowed. Calculated columns remain read-only in the live result set.

4) There is no GROUP BY clause.

5) There is no ORDER BY clause, or there is an ORDER BY clause that minimally matches an existing index in the source table in terms of fields (from left to right) and case-sensitivity.

6) There is no TOP N clause.

7) There are no sub-queries in the WHERE clause.

Multi-table queries

All queries that join two or more tables or union two or more SELECT statements will automatically produce a canned result set.

Calculated Columns

For live query result sets with calculated fields, additional internal information identifies a result column as both read-only and calculated. Every update of any column in a given row causes recalculation of any dependent calculated columns in that same row.

Identifying a Live Result Set

You may use the TDBISAMQuery ResultIsLive property to determine if the result set of a given SELECT SQL statement is live or canned after the query has been executed:

```
{
   MyQuery->SQL->Clear();
   MyQuery->SQL->Add("SELECT * FROM customer INNER JOIN");
   MyQuery->SQL->Add("INNER JOIN orders ON customer.ID=orders.CustID");
   MyQuery->SQL->Add("WHERE customer.ID=1000");
   MyQuery->Open();
   // In this case the result set will be canned due
   // to the join condition
   if (ResultIsLive)
      {
      ShowMessage("The result set is live");
      }
   else
      {
      ShowMessage("The result set is canned");
      }
}
```

Temporary Files

If a SELECT SQL statement is generating a canned result set, a temporary table will be created in the directory specified by the TDBISAMSession PrivateDir property for local sessions. If the query is being executed within a remote session, the location of the temporary table for the canned result set will be determined by the database server's configuration setting for the location of temporary tables, which can be modified remotely via the TDBISAMSession ModifyRemoteConfig method or locally on the server via the TDBISAMEngine ModifyServerConfig method. The TDBISAMQuery SessionName property determines what session is being used for the execution of the SQL statement. Please see the DBISAM Architecture topic for more information.

2.26 Parameterized Queries

Introduction

Parameters allow the same SQL statement to be used with different data values, and are placeholders for those data values. At runtime, the application prepares the query with the parameters and fills the parameter with a value before the query is executed. When the query is executed, the data values passed into the parameters are substituted for the parameter placeholder and the SQL statement is applied.

Specifying Parameters in SQL

Parameter markers can be used in SQL SELECT, INSERT, UPDATE, and DELETE statements in place of constants. Parameters are identified by a preceding colon (:). For example:

```
SELECT Last_Name, First_Name
FROM Customer
WHERE (Last_Name=:LName) AND (First_Name=:FName)
```

Parameters are used to pass data values to be used in WHERE clause comparisons and as update values in updating SQL statements such as UPDATE or INSERT. Parameters cannot be used to pass values for database, table, column, or index names. The following example uses the TotalParam parameter to pass the data value that needs to be assigned to the ItemsTotal column for the row with the OrderNo column equal to 1014:

```
UPDATE Orders
SET ItemsTotal = :TotalParam
WHERE (OrderNo = 1014)
```

Populating Parameters with the TDBISAMQuery Component

You can use the TDBISAMQuery Params property to populate the parameters in an SQL statement with data values. You may use two different methods of populating parameters using the Params property:

- By referencing each parameter by its index position in the available list of parameters
- By referencing each parameter by name using the ParamByName method

The following is an example of using the index positions of the parameters to populate the data values for an INSERT SQL statement:

```
{
   MyQuery->SQL->Clear();
   MyQuery->SQL->Add("INSERT INTO Country (Name, Capital, Population)");
   MyQuery->SQL->Add("VALUES (:Name, :Capital, :Population)");
   MyQuery->Params[0]->AsString="Lichtenstein";
   MyQuery->Params[1]->AsString="Vaduz";
   MyQuery->Params[2]->AsInteger=420000;
   MyQuery->ExecSQL();
}
```

The next block of code is an example of using the TDBISAMQuery ParamByName method in order to populate the data values for a SELECT SQL statement:

```
{
   MyQuery->SQL->Clear();
   MyQuery->SQL->Add("SELECT *");
   MyQuery->SQL->Add("FROM Orders");
   MyQuery->SQL->Add("WHERE CustID = :CustID");
   MyQuery->ParamByName("CustID")->AsFloat=1221;
   MyQuery->Open();
end;
```

Parameters and Multiple SQL Statements

If you have specified multiple SQL statements, or a script, in the SQL property and wish to execute these multiple SQL statements with different parameters, you can assign an event handler to the TDBISAMQuery OnGetParams event. The OnGetParams event is fired once before the execution of each SQL statement specified in the SQL property. In the event handler you would specify the parameters in the same way as you would for a single SQL statement described above. You can also use the SQLStatementType property to find out the type of SQL statement currently being executed and the Text property to examine the current SQL statement being executed.

Preparing Parameterized Queries

It is usually recommended that you manually prepare parameterized queries that you intend to execute many times with different parameter values. This can result in significant performance improvements since the process of preparing a query can be time-consuming. The following is an example of inserting 3 records with different values using a manually-prepared, parameterized query:

```
{
   MyQuery->SQL->Clear();
   MyQuery->SQL.Add("INSERT INTO Customer (CustNo, Company");
   MyQuery->SQL.Add("VALUES (:CustNo, :Company)");
   // Manually prepare the query
   MyQuery->Prepare();
   MyQuery->ParamByName("CustNo")->AsInteger=1000;
   MyQuery->ParamByName("Company")->AsString="Chocolates, Inc.";
   MyQuery->ExecSQL();
   MyQuery->ParamByName("CustNo")->AsInteger=2000;
   MyQuery->ParamByName("Company")->AsString="Flowers, Inc.";
   MyQuery->ExecSQL();
   MyQuery->ParamByName("CustNo")->AsInteger=3000;
   MyQuery->ParamByName("Company")->AsString="Candies, Inc.";
   MyQuery->ExecSQL();
}
```

Note
Manually preparing a script with multiple SQL statements does not result in any performance benefit since DBISAM still only prepares the first SQL statement in the script and must prepare each subsequent SQL statement before it is executed.

2.27 Navigating Tables and Query Result Sets

Introduction

Navigation of tables and query result sets is accomplished through several methods of the TDBISAMTable and TDBISAMQuery components. The basic navigational methods include the First, Next, Prior, Last, and MoveBy methods. The Bof and Eof properties indicate whether the record pointer is at the beginning or at the end of the table or query result set, respectively. These methods and properties are used together to navigate a table or query result set.

Moving to the First or Last Record

The First method moves to the first record in the table or query result set based upon the current index order. The Last method moves to the last record in the table or query result set based upon the current index order. The following example shows how to move to the first and last records in a table:

```
{
   MyTable->First();
   //  do something to the first record
   MyTable->Last();
   // do something to the last record
}
```

Skipping Records

The Next method moves to the next record in the table or query result set based upon the current index order. If the current record pointer is at the last record in the table or query result set, then calling the Next method will set the Eof property to True and the record pointer will stay on the last record. The Prior method moves to the previous record in the table or query result set based upon the current index order. If the current record pointer is at the first record in the table or query result set, then calling the Prior method will set the Bof property to True and the record pointer will stay on the first record. The following example shows how to use the First and Next methods along with the Eof property to loop through an entire table:

```
{
   MyTable->First();
   while (!MyTable->Eof)
      {
      MyTable->Next();
      }
}
```

The following example shows how to use the Last and Prior methods along with the Bof property to loop backwards through an entire table:

```
{
   MyTable->Last();
   while (!MyTable->Bof)
```

```
      {
      MyTable->Prior();
      }
}
```

Skipping Multiple Records

The MoveBy method accepts a positive or negative integer that represents the number of records to move by within the table or query result set. A positive integer indicates that the movement will be forward while a negative integer indicates that the movement will be backward. The return value of the MoveBy method is the number of records actually visited during the execution of the MoveBy method. If the record pointer hits the beginning of file or hits the end of file then the return value of the MoveBy method will be less than the desired number of records. The following example shows how to use the MoveBy method to loop through an entire table 10 records at a time:

```
{
   MyTable->First();
   while (!MyTable->Eof)
      {
      MyTable->MoveBy(10);
      }
}
```

2.28 Updating Tables and Query Result Sets

Introduction

Updating of tables and query result sets is accomplished through several methods of the TDBISAMTable and TDBISAMQuery components. The basic update methods include the Append, Insert, Edit, Delete, FieldByName, Post, and Cancel methods. The State property indicates whether the current table or query result set is in Append/Insert mode (dsInsert), Edit mode (dsEdit), or Browse mode (dsBrowse). These methods and properties are used together in order to update a table or query result set. Depending upon your needs, you may require additional methods to update BLOB fields within a given table or query result set, and information on how to use these methods are discussed at the end of this topic.

Note
For the rest of this topic, a table or query result set will be referred to as a dataset to reduce the amount of references to both. Also, it is important to note here that a query result set can be either "live" or "canned", which affects whether an update to a query result set appears in the actual table being queried or whether it is limited to the result set. Please see the Live Queries and Canned Queries topic for more information.

Adding a New Record

The Append and Insert methods allow you to begin the process of adding a record to the dataset. The only difference between these two methods is the Insert method will insert a blank record buffer at the current position in the dataset, and the Append method will add a blank record buffer at the end of the dataset. This record buffer does not exist in the physical datset until the record buffer is posted to the actual dataset using the Post method. If the Cancel method is called, then the record buffer and any updates to it will be discarded. Also, once the record buffer is posted using the Post method it will be positioned in the dataset according to the active index order, not according to where it was positioned due to the Insert or Append methods.

The FieldByName method can be used to reference a specific field for updating and accepts one parameter, the name of the field to reference. This method returns a TField object if the field name exists or an error if the field name does not exists. This TField object can be used to update the data for that field in the record buffer via properties such as
AsString, AsInteger, etc.

The following example shows how to use the Append method to add a record to a table with the following structure:

```
Field #    Name              DataType     Size
-----------------------------------------------
1          CustomerID        ftString     10
2          CustomerName      ftString     30
3          ContactName       ftString     30
4          Phone             ftString     10
5          Fax               ftString     10
6          EMail             ftString     30
7          LastSaleDate      ftDate       0
8          Notes             ftMemo       0
```

```
Index Name      Fields In Index      Options
-------------------------------------------
(none)          CustomerID           ixPrimary
```

```
{
   MyDBISAMDataSet->Append();  // State property will now reflect dsInsert
   MyDBISAMDataSet->FieldByName("CustomerID")->AsString="100";
   MyDBISAMDataSet->FieldByName("CustomerName")->AsString="The Hardware
      Store";
   MyDBISAMDataSet->FieldByName("ContactName")->AsString="Bob Smith";
   MyDBISAMDataSet->FieldByName("Phone")->AsString="5551212";
   MyDBISAMDataSet->FieldByName("Fax")->AsString="5551616";
   MyDBISAMDataSet->FieldByName("Email")->AsString=
     "bobs@thehardwarestore.com";
   MyDBISAMDataSet->Post();  // State property will now return to dsBrowse
}
```

If the record that is being posted violates a min/max or required constraint for the dataset then an EDBISAMEngineError exception will be raised with the appropriate error code. This will also occur if the record being posted will cause a key violation in either the primary index or a secondary index defined as unique. The error codes for a min/max constraint exception are 9730 (min) and 9731 (max) and are defined as DBISAM_MINVALERR and DBISAM_MAXVALERR in the dbisamcn unit (Delphi) or dbisamcn header file (C++). The error code for a required constraint exception is 9732 and is defined as DBISAM_REQDERR in the dbisamcn unit (Delphi) or dbisamcn header file (C++). The error code for a key violation exception is 9729 and is defined as DBISAM_KEYVIOL in the dbisamcn unit (Delphi) or dbisamcn header file (C++). Please see the Exception Handling and Errors and Appendix B - Error Codes and Messages topics for general information on exception handling in DBISAM.

You may use the OnPostError event to trap for any of these error conditions and display a message to the user. You can also use a try..except block to do the same, and the approach is very similar. The following shows how to use an OnPostError event handler to trap for a key violation error:

```
void __fastcall TMyForm::MyTablePostError(TDataSet *DataSet,
   EDatabaseError *E, TDataAction &Action)
{
   Action=daAbort;
   if (dynamic_cast<EDBISAMEngineError*>(E))
      {
      if (dynamic_cast<EDBISAMEngineError&>(*E)->ErrorCode==
          DBISAM_KEYVIOL)
         {
         ShowMessage("A record with the same key value(s) "+
                     "already exists, please change the "+
                     "record to make the value(s) unique "+
                     "and re-post the record");
         }
      else
         {
         ShowMessage(E->Message);
         }
      }
   else
      {
      ShowMessage(E->Message);
```

```
      }
}
```

Note
You will notice that the OnPostError event handler uses the more general EDatabaseError exception object for it's exception (E) parameter. Because of this, you must always first determine whether the exception object being passed is actually an EDBISAMEngineError before casting the exception object and trying to access specific properties such as the ErrorCode property. The EDBISAMEngineError object descends from the EDatabaseError object.

The following shows how to use a try..except block to trap for a key violation error:

```
{
   try
      {
      MyDBISAMDataSet->Append();  // State property will now reflect dsInsert
      MyDBISAMDataSet->FieldByName("CustomerID")->AsString="100";
      MyDBISAMDataSet->FieldByName("CustomerName")->AsString=
            "The Hardware Store";
      MyDBISAMDataSet->FieldByName("ContactName")->AsString="Bob Smith";
      MyDBISAMDataSet->FieldByName("Phone")->AsString="5551212";
      MyDBISAMDataSet->FieldByName("Fax")->AsString="5551616";
      MyDBISAMDataSet->FieldByName("Email")->AsString=
             "bobs@thehardwarestore.com";
      MyDBISAMDataSet->Post();  // State property will now return to dsBrowse
      }
   catch(const Exception &E)
      {
      if (dynamic_cast<EDBISAMEngineError*>(E))
         {
         if (dynamic_cast<EDBISAMEngineError&>(*E)->ErrorCode==
             DBISAM_KEYVIOL)
            {
            ShowMessage("A record with the same key value(s) "+
                        "already exists, please change the "+
                        "record to make the value(s) unique "+
                        "and re-post the record");
            }
         else
            {
            ShowMessage(E->Message);
            }
         }
      else
         {
         ShowMessage(E->Message);
         }
      }
}
```

Editing an Existing Record

The Edit method allows you to begin the process of editing an existing record in the dataset. DBISAM offers the choice of a pessimistic or optimistic locking protocol, which is configurable via the LockProtocol property for the TDBISAMSession assigned to the current dataset (see the SessionName property for more information on setting the session for a dataset). With the pessimistic locking protocol a record lock is obtained when the Edit method is called. As long as the record is being edited DBISAM will hold a record lock on that record, and will not release this lock until either the Post or Cancel methods is called. With the optimistic locking protocol a record lock is not obtained until the Post method is called, and never obtained if the Cancel method is called. This means that another user or session is capable of editing the record and posting the changes to the record before the Post method is called, thus potentially causing an EDBISAMEngineError exception to be triggered with the error code 8708, which indicates that the record has been changed since the Edit method was called and cannot be overwritten. In such a case you must discard the edited record by calling the Cancel method and begin again with a fresh copy of the record using the Edit method.

Note
Any updates to the record are done via a record buffer and do not actually exist in the actual dataset until the record is posted using the Post method. If the Cancel method is called, then any updates to the record will be discarded. Also, once the record is posted using the Post method it will be positioned in the dataset according to the active index order based upon any changes made to the record. What this means is that if any field that is part of the current active index is changed, then it is possible for the record to re-position itself in a completely different place in the dataset after the Post method is called.

The following example shows how to use the Edit method to update a record in a dataset:

```
{
   MyDBISAMDataSet->Edit();  // State property will now reflect dsEdit
   // Set LastSaleDate field to today's date
   MyDBISAMDataSet->FieldByName("LastSaleDate")->AsDateTime=Date;
   MyDBISAMDataSet->Post();  // State property will now return to dsBrowse
}
```

If the record that you are attempting to edit (or post, if using the optimistic locking protocol) is already locked by another user or session, then an EDBISAMEngineError exception will be triggered with the appropriate error code. The error code for a record lock error is 10258 and is defined as DBISAM_RECLOCKFAILED in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

It is also possible that the record that you are attempting to edit (or post) has been changed or deleted by another user or session since it was last cached by DBISAM. If this is the case then a DBISAM exception will be triggered with the error code 8708 which is defined as DBISAM_KEYORRECDELETED in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

You may use the OnEditError (or OnPostError, depending upon the locking protocol) event to trap for these error conditions and display a message to the user. You can also use a try..except block to do the same, and the approach is very similar. The following shows how to use an OnEditError event handler to trap for several errors:

```
void __fastcall TMyForm::MyTableEditError(TDataSet *DataSet,
    EDatabaseError *E, TDataAction &Action)
{
```

```
   Action=daAbort;
   if (dynamic_cast<EDBISAMEngineError*>(E))
      {
      if (dynamic_cast<EDBISAMEngineError&>(*E)->ErrorCode==
          DBISAM_RECLOCKFAILED)
         {
         if (MessageDlg("The record you are trying to edit "+
                        "is currently locked, do you want to "+
                        "try to edit this record again?",
                        mtWarning,TMsgDlgButtons()<<mbYes<<mbNo,0)==mrYes)
            {
            Action=daRetry;
            }
         }
      else if (dynamic_cast<EDBISAMEngineError&>(*E)->ErrorCode==
          DBISAM_KEYORRECDELETED)
         {
         MessageDlg("The record you are trying to edit "+
                    "has been modified since it was last "+
                    "retrieved, the record will now be "+
                    "refreshed",mtWarning,TMsgDlgButtons()<<mbOk,0);
         DataSet->Refresh;
         Action=daRetry;
         }
      else
         {
         MessageDlg(E.Message,mtError,TMsgDlgButtons()<<mbOk,0);
         }
      }
   else
      {
      MessageDlg(E.Message,mtError,TMsgDlgButtons()<<mbOk,0);
      }
}
```

The following shows how to use a try..except block to trap for several errors:

```
{
   while (true)
      {
      try
         {
         MyDBISAMDataSet->Edit();  // State property will now reflect dsEdit
         // Set LastSaleDate field to today's date
         MyDBISAMDataSet->FieldByName("LastSaleDate")->AsDateTime=Date;
         MyDBISAMDataSet->Post();
         // State property will now return to dsBrowse
         break; // Break out of retry loop
         }
      catch(const Exception &E)
         {
         if (dynamic_cast<EDBISAMEngineError*>(E))
            {
            if (dynamic_cast<EDBISAMEngineError&>(*E)->ErrorCode==
                DBISAM_RECLOCKFAILED)
               {
               if (MessageDlg("The record you are trying to edit "+
```

```
                              "is currently locked, do you want to "+
                              "try to edit this record again?",
                              mtWarning,TMsgDlgButtons()<<mbYes<<mbNo,
                              0)==mrYes)
                     {
                     continue;
                     }
                  }
               else if (dynamic_cast<EDBISAMEngineError&>(*E)->ErrorCode==
                        DBISAM_KEYORRECDELETED)
                  {
                  MessageDlg("The record you are trying to edit "+
                             "has been modified since it was last "+
                             "retrieved, the record will now be "+
                             "refreshed",mtWarning,TMsgDlgButtons()<<mbOk,0);
                  MyTable->Refresh();
                  continue;
                  }
               else
                  {
                  MessageDlg(E.Message,mtError,TMsgDlgButtons()<<mbOk,0);
                  break;
                  }
               }
            else
               {
               MessageDlg(E.Message,mtError,TMsgDlgButtons()<<mbOk,0);
               break;
               }
            }
         }
}
```

Deleting an Existing Record

The Delete method allows you to delete an existing record in a dataset. Unlike the Append, Insert, and Edit methods, the Delete method is a one-step process and does not require a call to the Post method to complete its operation. A record lock is obtained when the Delete method is called and is released as soon as the method completes. After the record is deleted the current position in the dataset will be the next closest record based upon the active index order.

The following example shows how to use the Delete method to delete a record in a dataset:

```
{
   MyDBISAMDataSet->Delete();
}
```

If the record that you are attempting to delete is already locked by another user or session, then an EDBISAMEngineError exception will be triggered with the appropriate error code. The error code for a record lock error is 10258 and is defined as DBISAM_RECLOCKFAILED in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

It is also possible that the record that you are attempting to delete has been changed or deleted by another user since it was last cached by DBISAM. If this is the case then an EDBISAMEngineError

exception will be triggered with the error code 8708 which is defined as DBISAM_KEYORRECDELETED in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

You may use the OnDeleteError event to trap for these error conditions and display a message to the user. You can also use a try..except block to do the same, and the approach is very similar. The code for an handling Delete errors is the same as that of an Edit, so please refer to the above code samples for handling Edit errors.

Cancelling an Insert/Append or Edit Operation

You may cancel an existing Insert/Append or Edit operation by calling the Cancel method. Doing this will discard any updates to an existing record if you are editing, or will completely discard a new record if you are inserting or appending. The following example shows how to cancel an edit operation on an existing record:

```
{
   MyDBISAMDataSet->Edit();  // State property will now reflect dsEdit
   // Set LastSaleDate field to today's date
   MyDBISAMDataSet->FieldByName("LastSaleDate")->AsDateTime=Date;
   MyDBISAMDataSet->Cancel();  // State property will now return to dsBrowse
}
```

Additional Events

There are several additional events that can be used to hook into the updating process for a dataset. They include the BeforeInsert, AfterInsert, OnNewRecord, BeforeEdit, AfterEdit, BeforeDelete, AfterDelete, BeforePost, AfterPost, BeforeCancel, and AfterCancel events. All of these events are fairly self-explanatory, however the OnNewRecord is special in that it can be used to assign values to fields in a newly-inserted or appended record without having the dataset mark the record as modified. If a record has not been modified in any manner, then the dataset will not perform an implicit Post operation when navigating off of the record. Instead, the Cancel method will be called and the record discarded.

Updating BLOB Fields

Most of the time you can simply use the general TField AsString and AsVariant properties to update a BLOB field in the same fashion as you would any other field. Both of these properties allow very large strings or binary data to be stored in a BLOB field. However, in certain cases you may want to take advantage of additional methods and functionality that are available through the TBlobField object that descends from TField or the TDBISAMBlobStream object that provides a stream interface to a BLOB field. The most interesting methods of the TBlobField object are the LoadFromFile, LoadFromStream, SaveToFile, and SaveToStream methods. These methods allow you to very easily load and save the data to and from BLOB fields.

Note
You must make sure that the dataset's State property is either dsInsert or dsEdit before using the LoadFromFile or LoadFromStream methods.

The following is an example of using the LoadFromFile method of the TBlobField object to load the contents of a text file into a memo field:

```
{
   MyDBISAMDataSet->Edit();  // State property will now reflect dsEdit
   // Load a text file from disk
   dynamic_cast<TBlobField&>(*MyDBISAMDataSet->FieldByName("Notes")->
          LoadFromFile("c:\\temp\\test.txt");
   MyDBISAMDataSet->Post(); // State property will now return to dsBrowse
}
```

Note
You'll notice that we must cast the result of the FieldByName method, which returns a TField object reference, to a TBlobField type in order to allow us to call the LoadFromFile method. This is okay since a memo field is a TMemoField object, which descends directly from TBlobField, which itself descends directly from TField.

In addition to these very useful methods, you can also directly manipulate a BLOB field like any other stream by using the TDBISAMBlobStream object. The following is an example of using a TDBISAMBlobStream component along with the TDBISAMTable or TDBISAMQuery SaveToStream method for storing DBISAM tables themselves in the BLOB field of another table:

```
{
   TDBISAMBlobStream *Stream;

   // First create the BLOB stream - be sure to make sure that
   // we put the table into dsEdit or dsInsert mode first since
   // we're writing to the BLOB stream
   MyFirstDBISAMDataSet->Append();
   try
      {
      MyBlobStream=new TDBISAMBlobStream((TBlobField *)
           MyFirstDBISAMDataSet->FieldByName("TableStream"),bmWrite);
      try
         {
         // Now save the table to the BLOB stream
         MySecondDBISAMDataSet->SaveToStream(MyBlobStream);
         }
      __finally
         {
         // Be sure to free the BLOB stream *before* the Post
         delete MyBlobStream;
         }
      MyFirstDBISAMDataSet->Post();
      }
   catch
      {
      // Cancel on an exception
      MyFirstDBISAMDataSet->Cancel();
      }
}
```

Note
For proper results when updating a BLOB field using a TDBISAMBlobStream object, you must create the TDBISAMBlobStream object after calling the Append/Insert or Edit methods for the dataset containing the BLOB field. Also, you must free the TDBISAMBlobStream object before calling the Post method to post the changes to the dataset. Finally, be sure to use the proper open mode when creating a TDBISAMBlobStream object for updating (either bmReadWrite or bmWrite).

2.29 Searching and Sorting Tables and Query Result Sets

Introduction

Searching and sorting tables and query result sets is accomplished through several methods of the TDBISAMTable and TDBISAMQuery components. The basic searching methods for tables (not query result sets) include the FindKey, FindNearest, SetKey, EditKey, GotoKey, and GotoNearest methods. The KeyFieldCount property is used with the SetKey and EditKey methods to control searching using the GotoKey and GotoNearest methods. The extended searching methods that do not necessarily rely upon an index and can be used with both tables and query result sets include the Locate, FindFirst, FindLast, FindNext, and FindPrior methods. The basic sorting methods for tables include the IndexName and IndexFieldNames properties.

Changing the Sort Order

You may use the IndexName and IndexFieldNames properties to set the current index order, and in effect, sort the current table based upon the index definition for the selected index order.

The IndexName property is used to set the name of the current index. For primary indexes, this property should always be set to blank (""). For secondary indexes, this property should be set to the name of the secondary index that you wish to use as the current index order. The following example shows how you would set the current index order for a table to a secondary index called "CustomerName":

```
{
   MyTable->IndexName="CustomerName";
   // do something
}
```

Note
Changing the index order can cause the current record pointer to move to a different position in the table (but not necessarily move off of the current record unless the record has been changed or deleted by another session). Call the First method after setting the IndexName property if you want to have the record pointer set to the beginning of the table based upon the next index order. Changing the index order will also remove any ranges that are active. Since the record numbers in DBISAM are based upon the index order the record number may also change.

If you attempt to set the IndexName property to a non-existent index an EDBISAMEngineError exception will be raised with the appropriate error code. The error code given for an invalid index name is 10022 and is defined as DBISAM_INVALIDINDEXNAME in the dbisamcn unit (Delphi) or dbisamcn header file (C++).

The IndexFieldNames property is used to set the current index order by specifying the field names of the desired index instead of the index name. Multiple field names should be separated with a semicolon. Using the IndexFieldNames property is desirable in cases where you are trying to set the current index order based upon a known set of fields and do not have any knowledge of the index names available. The IndexFieldNames property will attempt to match the given number of fields with the same number of beginning fields in any of the available primary or secondary indexes. The following example shows how you would set the current index order to a secondary index called "CustomerName" that consists of the CustomerName field and the CustomerNo field:

```
{
   MyTable->IndexFieldNames="CustomerName;CustomerNo";
   // do something
}
```

Note
Setting the IndexFieldNames will not work on indexes that are marked as descending or case-insensitive, so you must use the IndexName property instead. Also, if DBISAM cannot find any indexes that match the desired field names an EDatabaseError exception will be raised instead of an EDBISAMEngineError exception. If you are using this method of setting the current index order you should also be prepared to trap for this exception and deal with it appropriately.

Searching Using an Index

The FindKey method accepts an array of search values to use in order to perform an exact search for a given record using the active index. The return value of the FindKey method indicates whether the search was successful. If the search was successful then the record pointer is moved to the desired record, whereas if the search was not successful then the record pointer stays at its current position. The search values must correspond to the fields that make up the active index or the search will not work properly. However, FindKey does not require that you fill in all of the field values for all of the fields in the active index, rather only that you fill in the field values from left to right. The following example shows how to perform a search on the primary index comprised of the CustomerNo field:

```
{
   // Set to the primary index
   MyTable->IndexName="";
   // Search for customer 100
   // With C++, the field values must be passed
   // as either a TVarRec (for single values) or an
   // ARRAYOFCONST
   TVarRec SearchValue=(100);
   if (MyTable->FindKey(&SearchValue))
      {
      // Record was found, now do something
      }
   else
      {
      ShowMessage("Record was not found");
      }
}
```

The FindNearest method accepts an array of search values to use in order to perform a near search for a given record using the active index. If the search was successful then the record pointer is moved to the desired record, whereas if the search was not successful then the record pointer is moved to the next record that most closely matches the current search values. If there are no records that are greater than the search values then the record pointer will be positioned at the end of the table. The search values must correspond to the fields that make up the active index or the search will not work properly. However, FindNearest does not require that you fill in all of the field values for all of the fields in the active index, rather only that you fill in the field values from left to right. The following example shows how to perform a near search on the primary index comprised of the CustomerNo field:

```
{
   // Set to the primary index
   MyTable->IndexName="";
   // Search for customer 100 or nearest
   // With C++, the field values must be passed
   // as either a TVarRec (for single values) or an
   // ARRAYOFCONST
   TVarRec SearchValue=(100);
   MyTable->FindNearest(&SearchValue);
}
```

The SetKey and EditKey methods are used in conjunction with the GotoKey and GotoNearest methods to perform searching using field assignments instead of an array of field values. The SetKey method begins the search process by putting the TDBISAMTable component into the dsSetKey state and clearing all field values. You can examine the state of the table using the State property. The application must then assign values to the desired fields and call the GotoKey or GotoNearest method to perform the actual search. The GotoNearest method may be used if you wish to perform a near search instead of an exact search. The EditKey method extends or continues the current search process by putting the TDBISAMTable component into the dsSetKey state but not clearing any field values. This allows you to change only one field without being forced to re-enter all field values needed for the search. The KeyFieldCount property controls how many fields, based upon the current index, are to be used in the actual search. By default the KeyFieldCount property is set to the number of fields for the active index. The following example shows how to perform an exact search using the SetKey and GotoKey methods and KeyFieldCount property. The active index is a secondary index called "CustomerName" comprised of the CustomerName field and the CustomerNo field:

```
{
   // Set to the CustomerName secondary index
   MyTable->IndexName="CustomerName";
   // Search for the customer with the
   // name "The Hardware Store"
   MyTable->SetKey();
   MyTable->FieldByName("CustomerName")->AsString="The Hardware Store";
   // This causes the search to only look at the first field
   // in the current index when searching
   MyTable->KeyFieldCount=1;
   if (MyTable->GotoKey())
      {
      // Record was found, now do something
      }
   else
      {
      ShowMessage("Record was not found");
      }
}
```

Note
In the previous example we executed a partial-field search. What this means is that we did not include all of the fields in the active index. DBISAM does not require that you use all of the fields in the active index for searching.

The following example shows how to perform a near search using the SetKey and GotoNearest methods, and KeyFieldCount property. The active index is a secondary index called "CustomerName" comprised of the CustomerName field and the CustomerNo field:

```
{
   // Set to the CustomerName secondary index
   MyTable->IndexName="CustomerName";
   // Search for the customer with the
   // name "The Hardware Store"
   MyTable->SetKey();
   MyTable->FieldByName("CustomerName")->AsString="The Hardware Store";
   // This causes the search to only look at the first field
   // in the current index when searching
   MyTable->KeyFieldCount=1;
   MyTable->GotoNearest();
}
```

Searching Without a Specific Index Order Set

The Locate method is used to locate a record independent of the active index order or of any indexes at all. This is why it can be used with query result sets in addition to tables. The Locate method will attempt to use the active index for searching, but if the current search fields do not match the active index then the Locate method will attempt to use another available index. Indexes are selected based upon the options passed to the Locate method in conjunction with the field names that you wish to search upon. The index fields are checked from left to right, and if a primary or secondary index is found that matches the search fields from left to right and satisfies the options desired for the search it will be used to perform the search. Finally, if no indexes can be found that can be used for the search, a filter will be used to execute the search instead. This is usually a sub-optimal solution and can take a bit of time since the filter will be completely un-optimized and will be forced to scan every record for the desired field values.

The Locate method accepts a list of field names as its first argument. Multiple field names should be separated with a semicolon. These are the fields you wish to search on. The second argument to the Locate method is an array of field values that should correspond to the field names passed in the first argument. The third and final argument is a set of options for the Locate method. These options control how the search is performed and how indexes are selected in order to perform the search. The return value of the Locate method indicates whether the current search was successful. If the search was successful then the record pointer is moved to the desired record, whereas if the search was not successful then the record pointer stays at its current position.

The following example shows how to use the Locate method to find a record where the CustomerName field is equal to "The Hardware Store":

```
{
   Variant SearchValues[1];
   SearchValues[0]=Variant("The Hardware Store");
   // Search for the customer with the
   // name "The Hardware Store"
   if (MyTable->Locate("CustomerName",
                       VarArrayOf(SearchValues,2),
                       TLocateOptions()))
      {
      // Record was found, now do something
```

```
      }
   else
      {
      ShowMessage("Record was not found");
      }
}
```

The following example shows how to use the Locate method to find a record where the CustomerName field is equal to "The Hardware Store", but this time the search will be case-insensitive:

```
{
   Variant SearchValues[1];
   SearchValues[0]=Variant("The Hardware Store");
   // Search for the customer with the
   // name "The Hardware Store"
   if (MyTable->Locate("CustomerName",
                       VarArrayOf(SearchValues,2),
                       TLocateOptions() << loCaseInsensitive))
      {
      // Record was found, now do something
      }
   else
      {
      ShowMessage("Record was not found");
      }
}
```

The FindFirst, FindLast, FindNext, and FindPrior methods all rely on the Filter and FilterOptions properties to do their work. These methods are the most flexible for searching and can be used with both tables and query result sets, but there are some important caveats. To get acceptable performance from these methods you must make sure that the filter expression being used for the Filter property is optimized or at least partially-optimized. If the filter expression is un-optimized it will take a significantly greater amount of time to complete every call to any of the FindFirst, FindLast, FindNext, or FindPrior methods unless the table or query result set being searched only has a small number of records. Please see the Filter Optimization topic for more information. Also, because the Filter property is being used for these methods, you cannot use a different filter expression in combination with these methods. However, you can set the Filtered property to True and show only the filtered records if you so desire. Finally, the FilterOptions property controls how the filtering is performed during the searching, so you should make sure that these options are set properly. The following example shows how to use the Filter property and FindFirst and FindNext methods to find matching records and navigate through them in a table:

```
{
   // Search for the first customer with the
   // name "The Hardware Store"
   MyTable->Filter="CustomerName="+QuotedStr("The Hardware Store");
   // We want the search to be case-insensitive
   TFilterOptions FilterOptions;
   FilterOptions->Clear();
   FilterOptions << foCaseInsensitive;
   MyTable->FilterOptions=FilterOptions;
   if (MyTable->FindFirst())
      {
      // Record was found, now search through
      // the rest of the matching records
```

```
      while (FindNext())
         {
         // Do something here
         }
      }
   else
      {
      ShowMessage("Record was not found");
      }
}
```

2.30 Setting Ranges on Tables

Introduction

Setting ranges on tables is accomplished through several methods of the TDBISAMTable component. The basic range methods include the SetRange, SetRangeStart, SetRangeEnd, EditRangeStart, EditRangeEnd, and ApplyRange methods. The KeyFieldCount property is used with the SetRangeStart, SetRangeEnd, EditRangeStart and EditRangeEnd methods to control searching using the ApplyRange method. All range operations are dependent upon the active index order set using the IndexName or IndexFieldNames properties. Ranges may be combined with expression filters set using the Filter and Filtered propertes and/or callback filters set using the OnFilterRecord event to further filter the records in the table.

Setting a Range

The SetRange method accepts two arrays of values to use in order to set a range on a given table. If the current record pointer does not fall into the range values specified, then the current record pointer will be moved to the nearest record that falls within the range. These value arrays must contain the field values in the same order as the field names in the active index or the range will not return the desired results. However, SetRange does not require that you fill in all of the field values for all of the fields in the active index, rather only that you fill in the field values from left to right. The following example shows how to perform a range on the primary index comprised of the CustomerNo field:

```
{
   // Set to the primary index
   MyTable->IndexName="";
   // Set a range from customer 100 to customer 300
   MyTable->SetRange(ARRAYOFCONST((100)),
                     ARRAYOFCONST((300)));
}
```

The SetRangeStart, SetRangeEnd, EditRangeStart, and EditRangeEnd methods are used in conjunction with the ApplyRange method to perform a range using field assignments instead of arrays of field values. The SetRangeStart method begins the range process by putting the TDBISAMTable component into the dsSetKey state and clearing all field values. You can examine the state of the table using the State property. The application must then assign values to the desired fields for the start of the range and then proceed to call SetRangeEnd to assign values to the desired fields for the end of the range. After this is done the application can call the ApplyRange method to perform the actual range operation. The EditRangeStart and EditRangeEnd methods extend or continue the current range process by putting the TDBISAMTable component into the dsSetKey state but not clearing any field values. You can examine the state of the table using the State property. This allows you to change only one field without being forced to re-enter all field values needed for the beginning or ending values of the range. The KeyFieldCount property controls how many fields, based upon the active index, are to be used in the actual range and can be set independently for both the starting and ending field values of the range. By default the KeyFieldCount property is set to the number of fields in the active index. The following example shows how to perform a range using the SetRangeStart, SetRangeEnd, and ApplyRange methods and KeyFieldCount property. The active index is a secondary index called "CustomerName" that consists of the CustomerName field and the CustomerNo field:

```
{
   // Set to the CustomerName secondary index
```

```
   MyTable->IndexName="CustomerName;
   // Set a range to find all customers with
   // a name beginning with 'A'
   MyTable->SetRangeStart();
   MyTable->FieldByName("CustomerName")->AsString="A";
   // This causes the range to only look at
   // the first field in the current index
   MyTable->KeyFieldCount=1;
   MyTable->SetRangeEnd();
   // Note the padding of the ending range
   // values with lowercase z's
   // to the length of the CustomerName
   // field, which is 20 characters
   MyTable->FieldByName("CustomerName")->
            AsString="Azzzzzzzzzzzzzzzzzz";
   // This causes the range to only look at
   // the first field in the current index
   MyTable->KeyFieldCount=1;
   MyTable->ApplyRange();
}
```

Note
In the previous example we executed a partial-field range. What this means is that we did not include all of the fields in the active index in the range. DBISAM does not require that you use all of the fields in the active index for the range.

2.31 Setting Master-Detail Links on Tables

Introduction

A master-detail link is a property-based linkage between a master TDataSource component and a detail TDBISAMTable component. Once a master-detail link is established, any changes to the master TDataSource component will cause the detail TDBISAMTable component to automatically reflect the change and show only the detail records that match the current master record based upon the link criteria. Master-detail links use ranges for their functionality, and therefore are dependent upon the active index in the detail table. Like ranges, master-detail links may be combined with expression filters set using the Filter and Filtered propertes and/or callback filters set using the OnFilterRecord event to further filter the records in the detail table.

Defining the Link Properties

Setting master-detail links on tables is accomplished through four properties in the detail TDBISAMTable component. These properties are the MasterSource, MasterFields, IndexName, and IndexFieldNames properties.

The first step in setting a master-detail link is to assign the MasterSource property. The MasterSource property refers to a TDataSource component. This makes master-detail links very flexible, because the TDataSource component can provide data from any TDataSet-descendant component such as a TDBISAMTable or TDBISAMQuery component as well as many other non-DBISAM dataset components.

> **Note**
> For the link to be valid, the TDataSource DataSet property must refer to a valid TDataSet-descendant component.

The next step is to assign the IndexName property, or IndexFieldNames property, so that the active index, and the fields that make up that index, will match the fields that you wish to use for the link. The only difference between specifying the IndexName property versus the IndexFieldNames property is that the IndexName property expects the name of an index, whereas the IndexFieldNames only expects the names of fields in the table that match the fields found in an index in the table from left-to-right. The IndexFieldNames property also does not require that all of the fields in an existing index be specified in order to match with that existing index, only enough to be able to select the index so that it will satisfy the needs of the master-detail link.

Finally, the MasterFields property must be assigned a value. This property requires a field or list of fields separated by semicolons from the master data source that match the fields in the active index for the detail table.

To illustrate all of this we'll use an example. Let's suppose that we have two tables with the following structure and we wish to link them via a master-detail link:

```
Customer Table

Field #    Name              DataType     Size
-----------------------------------------------
1          CustomerID        ftString     10
2          CustomerName      ftString     30
```

```
3          ContactName       ftString     30
4          Phone             ftString     10
5          Fax               ftString     10
6          EMail             ftString     30
```

Note
Indexes in this case are not important since this will be the master table

```
Orders Table

Field #    Name              DataType     Size
-----------------------------------------------
1          CustomerID        ftString     10
2          OrderNumber       ftString     10
3          OrderDate         ftDate       0
4          OrderAmount       ftBCD        2


Index Name      Fields In Index             Options
-----------------------------------------------------
(none)          CustomerID;OrderNumber      ixPrimary
```

We would use the following example code to establish a master-detail link between the two tables. In this example it is assumed that a TDataSource component called CustomerSource exists and points to a TDBISAMTable component for the "customer" table:

```
{
   // Select the primary index, which contains the
   // CustomerID and OrderNumber fields
   OrdersTable->IndexName="";
   // Assign the MasterSource property
   OrdersTable->MasterSource=CustomerSource;
   // Set the MasterFields property to point to the
   // CustomerID field from the Customer table
   OrdersTable->MasterFields="CustomerID";
}
```

Now any time the current record in the CustomerSource data source changes in any way, the OrdersTable will automatically reflect that change and only show records that match the master record's CustomerID field. Below is the same example, but changed to use the IndexFieldNames property instead:

```
{
   // Select the primary index, which contains the
   // CustomerID and OrderNumber fields
   OrdersTable->IndexFieldNames="CustomerID";
   // Assign the MasterSource property
   OrdersTable->MasterSource=CustomerSource;
   // Set the MasterFields property to point to the
   // CustomerID field from the Customer table
```

```
    OrdersTable->MasterFields="CustomerID";
}
```

Note
Because a master-detail link uses data-event notification in the TDataSource component for maintaining the link, if the TDataSet component referred to by the TDataSource component's DataSet property calls its DisableControls method, it will not only disable the updating of any data-aware controls that refer to it, but it will also disable any master-detail links that refer to it also. This is the way the TDataSet and TDataSource components have been designed, so this is an expected behavior that you should keep in mind when designing your application.

2.32 Setting Filters on Tables and Query Result Sets

Introduction

Setting filters on tables and query result sets is accomplished through several properties of the TDBISAMTable and TDBISAMQuery components. These properties include the Filter, FilterOptions, Filtered, and FilterOptimizeLevel properties. The OnFilterRecord event is used to assign a callback filter event handler that can be used to filter records using Delphi or C++ code. All filter operations are completely independent of any active index order.

Setting an Expression Filter

The Filter, FilterOptions, Filtered, and FilterOptimizeLevel properties are used to set an expression filter. The steps to set an expression filter include setting the filter expression using the Filter property, specifying any filter options using the FilterOptions property, and then making the expression filter active by setting the Filtered property to True. You can turn off or disable an expression filter by setting the Filtered property to False. If the current record pointer does not fall into the conditions specified by an expression filter, then the current record pointer will be moved to the nearest record that falls within the filtered set of records. Expression filters may be combined with ranges, master-detail links, and/or callback filters to further filter the records in the table or query result set.

DBISAM's expression filters use the same naming conventions, operators, and functions as its SQL implementation. The only differences are as follows:

Difference	Description
Correlation Names	You cannot use table or column correlation names in filter expressions.
Aggregate functions	You cannot use any aggregate functions like SUM(), COUNT(), AVG(), etc. in filter expressions.

Please see the Naming Conventions, Operators, and Functions topics in the SQL Reference for more information.

Note
Unlike with SQL, you may also use use the asterisk (*) character to specify a partial-length match for string field comparisons in a filter expression. However, this only works when the foNoPartialCompare element is not included in the FilterOptions property.

The following example shows how to set an expression filter where the LastSaleDate field is between January 1, 1998 and December 31, 1998 and the TotalSales field is greater than 10,000 dollars:

```
{
   // Set the filter expression
   MyTable->Filter="(LastSaleDate >= "+QuotedStr("1998-01-01")+") "+
             "and (LastSaleDate <= "+QuotedStr("1998-12-31")+") "+
             "and (TotalSales > 10000)';
   TFilterOptions FilterOptions;
   FilterOptions->Clear();
```

```
   MyTable->FilterOptions=FilterOptions;
   MyTable->Filtered=true;
}
```

DBISAM attempts to optimize all expression filters. This means that DBISAM will try to use existing indexes to speed up the filter operation. The FilterOptimizeLevel property indicates what level of optimization was, or will be, achieved for the expression filter and can be examined after the Filtered property is set to True to execute the filter. The following example displays a message dialog indicating the level of optimization achieved for the expression filter:

```
{
   // Set the filter expression, in this case for
   // a partial-match, case-insensitive filter
   MyTable->Filter="CustomerName = "+QuotedStr("A*");
   TFilterOptions FilterOptions;
   FilterOptions->Clear();
   FilterOptions << foCaseInsensitive;
   MyTable->FilterOptions=FilterOptions;
   MyTable->Filtered=true;
   switch (MyTable->FilterOptimizeLevel)
      {
      case foNone:
         {
         ShowMessage("The filter is completely unoptimized");
         }
      case foPartial:
         {
         ShowMessage("The filter is partially optimized");
         }
      case foFull:
         {
         ShowMessage("The filter is completely optimized");
         }
      }
}
```

Note
The foCaseInsensitive filter option can affect the optimization level returned by the FilterOptimizeLevel, so you should make sure to set any filter options before examining the FilterOptimizeLevel property so as to avoid any confusion.

Please see the Filter Optimization topic for more information.

Setting a Callback Filter

The OnFilterRecord event and the Filtered property are used together to set a callback filter. The steps to set a callback filter include assigning an event handler to the OnFilterRecord event and then making the callback filter active by setting the Filtered property to True. You can turn off or disable a callback filter by setting the Filtered property to False. If the current record pointer does not fall into the conditions specified within the callback filter, then the current record pointer will be moved to the nearest record that falls within the filtered set of records.

The following example shows how to write a callback filter event handler where the CustomerName field contains the word "Hardware" (case-sensitive):

```
void __fastcall TMyForm::TableFilterRecord(TDataSet *DataSet,
   bool &Accept);
{
   Accept=false;
   if (Pos("Hardware",
         DataSet->FieldByName("CustomerName")->AsString) > 0))
      {
      Accept=true;
      }
}
```

Note
Callback filters implemented via the OnFilterRecord event are always completely un-optimized. In order to satisfy the filter requirements, DBISAM must always read every record to determine if the record falls into the desired set of records. You should only use OnFilterRecord on small sets of data, or large sets of data that have been reduced to a small number of records by an existing range and/or expression filter.

2.33 Loading and Saving Streams with Tables and Query Result Sets

Introduction

Loading and saving tables and query result sets to and from streams is accomplished through the LoadFromStream and SaveToStream methods of the TDBISAMTable and TDBISAMQuery components. The properties used by the LoadFromStream and SaveToStream methods include the DatabaseName, TableName, and Exists properties. A stream is any TStream-descendant object such as TFileStream, TMemoryStream, or even the DBISAM TDBISAMBlobStream object used for reading and writing to BLOB fields. Loading a stream copies the entire contents of a stream to an existing table or query result set. When loading a stream, the contents of the stream must have been created using the SaveToStream method or else an EDBISAMEngineError exception will be raised. The error code given when a load from a stream fails because of an invalid stream is 11312 and is defined as DBISAM_LOADSTREAMERROR in the dbisamcn unit (Delphi) or dbisamcn header file (C++). Saving to a stream copies the contents of a table or query result set to the stream, overwriting the entire contents of the stream. The records that are copied can be controlled by setting a range or filter on the source table or query result set prior to calling the SaveToStream method. Please see the Setting Ranges on Tables and Setting Filters on Tables and Query Result Sets topics for more information.

Loading Data from a Stream

To load data from a stream into an existing table, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the LoadFromStream method. When using a TDBISAMTable component, the table can be open or closed when this method is called, and the table does not need to be opened exclusively. If the table is closed when this method is called, then DBISAM will attempt to open the table before loading the data into it. It is usually good practice to examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to load data into a non-existent table. If you do attempt to load data into a non-existent table an EDBISAMEngineError exception will be raised. The error code given when a load from a stream fails due to the table not existing is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++). To load data from a stream into a query result set, the TDBISAMQuery SQL property must be populated with a SELECT SQL statement and the Active property must be True.

The following example shows how to load data from a memory stream (assumed to already be created) into a table using the LoadFromStream method:

```
{
   MyTable->DatabaseName="d:\\temp";
   MyTable->TableName="customer";
   if (MyTable->Exists)
      {
      MyTable->LoadFromStream(MyMemoryStream);
      }
}
```

Note
Tables or query result sets in remote sessions can load streams from a local stream. However, since the stream contents are sent as one buffer to the database server as part of the request, it is recommended that you do not load particularly large streams since you will run the risk of exceeding the available memory on the local workstation or database server.

Tracking the Load Progress

To take care of tracking the progress of the load we have provided the TDBISAMTable and TDBISAMQuery OnLoadFromStreamProgress events.

Saving Data to a Stream

To save the data from a table to a stream, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the SaveToStream method. When using a TDBISAMTable component, the table can be open or closed when this method is called, and the table does not need to be opened exclusively. If the table is closed when this method is called, then DBISAM will attempt to open the table before saving the data. It is usually good practice to examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to save data from a non-existent table. If you do attempt to save data from a non-existent table an EDBISAMEngineError exception will be raised. The error code given when a save fails due to the table not existing is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++). To save data to a stream from a query result set, the TDBISAMQuery SQL property must be populated with a SELECT SQL statement and the Active property must be True.

The following example shows how to save the data from a table to a memory stream (assumed to already be created) using the SaveToStream method of the TDBISAMTable component:

```
{
   MyTable->DatabaseName="d:\\temp";
   MyTable->TableName="customer";
   if (MyTable->Exists)
      {
      MyTable->SaveToStream(MyMemoryStream);
      }
}
```

Tracking the Save Progress

To take care of tracking the progress of the save we have provided the TDBISAMTable and TDBISAMQuery OnSaveToStreamProgress events.

2.34 Importing and Exporting Tables and Query Result Sets

Introduction

Importing and exporting tables and query result sets to and from delimited text files is accomplished through the ImportTable and ExportTable methods of the TDBISAMTable and TDBISAMQuery components. The properties used by the ImportTable and ExportTable methods include the DatabaseName, TableName, and Exists properties. Importing a table copies the entire contents of a delimited text file to an existing table or query result set. Exporting a table copies the contents of a table or query result set to a new delimited text file. The records that are copied can be controlled by setting a range or filter on the source table or query result set prior to calling the ExportTable method. Please see the Setting Ranges on Tables and Setting Filters on Tables and Query Result Sets topics for more information.

Importing Data

To import a delimited text file into an existing table, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the ImportTable method. When using a TDBISAMTable component, the table can be open or closed when this method is called, and the table does not need to be opened exclusively. If the table is closed when this method is called, then DBISAM will attempt to open the table before importing the data into it. It is usually good practice to examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to import data into a non-existent table. If you do attempt to import data into a non-existent table an EDBISAMEngineError exception will be raised. The error code given when an import fails due to the table not existing is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++). To import a delimited text file into a query result set, the TDBISAMQuery SQL property must be populated with a SELECT SQL statement and the Active property must be True.

The following example shows how to import a delimited text file into a table using the ImportTable method:

```
Incoming text file has following layout:

Field #    Name              DataType
-------------------------------------
1          CustomerName      ftString
2          ContactName       ftString
3          Phone             ftString
4          Fax               ftString
5          EMail             ftString

Table has following structure:

Field #    Name              DataType     Size
-----------------------------------------------
1          CustomerID        ftAutoInc    0
2          CustomerName      ftString     30
3          ContactName       ftString     30
4          Phone             ftString     10
5          Fax               ftString     10
6          EMail             ftString     30
7          LastSaleDate      ftDate       0

Index Name      Fields In Index      Options
-----------------------------------------------
```

```
(none)          CustomerID          ixPrimary
```

```
// In this example we'll use a comma as a delimiter

{
   TStringList *IncomingFields=new TStringList;
   try
      {
      MyTable->DatabaseName="d:\\temp";
      MyTable->TableName="customer";
      if (MyTable->Exists)
         {
         IncomingFields->Add("CustomerName");
         IncomingFields->Add("ContactName");
         IncomingFields->Add("Phone");
         IncomingFields->Add("Fax");
         IncomingFields->Add("Email");
         // Date, time, and number formatting left
         // to defaults for this example
         MyTable->ImportTable("d:\\incoming\\customer.txt",
                              ",",false,IncomingFields);
         }
      }
   __finally
      {
      delete IncomingFields;
      }
}
```

Note
Tables or query result sets in remote sessions can only import delimited text files that are accessible from the database server on which the tables or query result sets reside. You must specify the path to the incoming text file in a form that the database server can use to open the file.

In addition to using the TDBISAMTable and TDBISAMQuery ImportTable methods for importing delimited text files, DBISAM also allows the use of the IMPORT TABLE SQL statement.

Tracking the Import Progress

To take care of tracking the progress of the import we have provided the TDBISAMTable and TDBISAMQuery OnImportProgress events.

Exporting Data

To export a table to a delimited text file, you must specify the DatabaseName and TableName properties of the TDBISAMTable component and then call the ExportTable method. When using a TDBISAMTable component, the table can be open or closed when this method is called, and the table does not need to be opened exclusively. If the table is closed when this method is called, then DBISAM will attempt to open the table before exporting the data. It is usually good practice to examine the Exists property of the TDBISAMTable component first to make sure that you don't attempt to export data from a non-existent table. If you do attempt to export data from a non-existent table an EDBISAMEngineError exception will be

raised. The error code given when an export fails due to the table not existing is 11010 and is defined as DBISAM_OSENOENT in the dbisamcn unit (Delphi) or dbisamcn header file (C++). To export data to a delimited text file from a query result set, the TDBISAMQuery SQL property must be populated with a SELECT SQL statement and the Active property must be True.

The following example shows how to export a table to a delimited text file using the ExportTable method of the TDBISAMTable component:

```
Outgoing text file should have the following layout:

Field #    Name              DataType
-------------------------------------
1          CustomerName      ftString
2          ContactName       ftString
3          Phone             ftString
4          Fax               ftString
5          EMail             ftString

Table has following structure:

Field #    Name              DataType     Size
---------------------------------------------
1          CustomerID        ftAutoInc    0
2          CustomerName      ftString     30
3          ContactName       ftString     30
4          Phone             ftString     10
5          Fax               ftString     10
6          EMail             ftString     30
7          LastSaleDate      ftDate       0

Index Name      Fields In Index      Options
-----------------------------------------------
(none)          CustomerID           ixPrimary
```

```
// In this example we'll use a comma as a delimiter
and only export records that have a non-blank email address
   TStringList *OutgoingFields=new TStringList;
   try
      {
      MyTable->DatabaseName="d:\\temp";
      MyTable->TableName="customer";
      if (MyTable->Exists)
         {
         MyTable->Open();
         try
            {
            MyTable->Filter="EMail IS NOT NULL";
            MyTable->Filtered=true;
            OutgoingFields->Add("CustomerName");
            OutgoingFields->Add("ContactName");
            OutgoingFields->Add("Phone");
            OutgoingFields->Add("Fax");
            OutgoingFields->Add("Email");
            // Date, time, and number formatting left
            // to defaults for this example
```

```
            MyTable->ExportTable("d:\\outgoing\\customer.txt",
                                 ",",false,OutgoingFields);
            }
          __finally
            {
            MyTable->Close();
            }
          }
       }
    __finally
       {
       delete OutgoingFields;
       }
}
```

Note
Tables or query result sets in remote sessions can only export data to delimited text files that are accessible from the database server on which the source tables or query result sets reside. You must specify the path to the text file in a form that the database server can use to create the file.

In addition to using the TDBISAMTable and TDBISAMQuery ExportTable methods for exporting data to delimited text files, DBISAM also allows the use of the EXPORT TABLE SQL statement.

Tracking the Export Progress

To take care of tracking the progress of the export we have provided the TDBISAMTable and TDBISAMQuery OnExportProgress events.

2.35 Cached Updates

Introduction

Using cached updates for table and query result sets is accomplished through the BeginCachedUpdates, and ApplyCachedUpdates, and CancelCachedUpdates methods of the TDBISAMTable and TDBISAMQuery components. The properties used by these methods include the CachingUpdates property. Using cached updates permits an application to copy all existing records in a given table or query result set to a temporary table that is then used for any inserts, updates, or deletes. Once all updates are complete, the application may then call the ApplyCachedUpdates method to apply all updates to the source table or query result set, or the CancelCachedUpdates method to cancel all updates and revert the table or query result set to its original state prior to the cached updates. The records that are included in the cached updates can be controlled by setting a range or filter on the source table or query result set prior to calling the BeginCachedUpdates method. Please see the Setting Ranges on Tables and Setting Filters on Tables and Query Result Sets topics for more information.

Note
Do not use cached updates on very tables or query result sets with large number of records in the active set according to any active ranges and/or filters. Doing so can result in some serious performance problems as the entire set of records will need to be copied when cached updates are begun.

Beginning Cached Updates

To begin cached updates, you must call the BeginCachedUpdates method. When using either a TDBISAMTable or TDBISAMQuery component, the table or query result set must be opened (Active property is set to True) or an exception will be raised.

Applying Cached Updates

To apply any cached updates to the source table or query result set, you must call the ApplyCachedUpdates method. This method will apply any updates that were made to the temporary table used for the cached updates to the source table or query result set. Only records that were inserted, updated, or deleted are processed, so the result is the same as calling the CancelCachedUpdates method if no records were inserted, updated, or deleted while cached updates were enabled. You can examine the CachingUpdates property to determine whether cached udpdates are in effect before trying to apply any cached updates.

It is strongly recommend that you always wrap the ApplyCachedUpdates method with a TDBISAMDatabase StartTransaction and Commit and Rollback block of code. This will allow the application of the cached updates to behave as an atomic unit of work and will avoid any possible problems of partial updates due to errors during the application of the updates.

The following example shows how to propery apply cached updates using a transaction around the ApplyCachedUpdates method:

```
{
   TStringList *TablesList=new TStringList;
   try
```

```
      {
      TablesList->Add(MyTable->TableName);
      MyTable->Database->StartTransaction(TablesList);
      try
         {
         MyTable->ApplyCachedUpdates();
         MyTable->Database->Commit();
         }
      catch
         {
         MyTable->Database->Rollback();
         throw;
         }
      }
   __finally
      {
      delete TablesList;
      }
}
```

Note
Notice that a restricted transaction is used in this example. It is wise to do this if only updating one table because it helps increase multi-user concurrency. Please see the Transactions topic for more information.

Reconciling Errors

Cached updates are handled in an optimistic manner, which means that DBISAM does not hold any locks on the records that are held in the cache while the cached updates are in effect. Subsequently, it is possible that another session has changed some or all of the records that were cached and updated or deleted in the cache. When the cached updates are then applied using the ApplyCachedUpdates method, an error message will be raised and it is possible that only a portion of the cached updates will be applied to the source table or query result set. To avoid this, you can assign an event handler to the OnCachedUpdateError event. This will cause DBISAM to instead call this event handler when an error occurs during the application of the cached updates, giving the user an opportunity to correct any errors and retry any update that is causing an error.

Note
No matter what happens with respect to errors, the ApplyCachedUpdates method always results in cached updates being turned off and the source table or query result being returned to its normal state.

The following is an example of an OnCachedUpdateError event handler that retries the current record application if a record lock error is causing the problem:

```
void __fastcall TMyForm::MyTableCachedUpdateError(TObject *Sender,
  TDBISAMRecord *CurrentRecord, Exception *E,
  TUpdateType UpdateType, TUpdateAction &Action)
{
   Action=uaFail;
```

```
    if (dynamic_cast<EDBISAMEngineError*>(E))
       {
       if (dynamic_cast<EDBISAMEngineError&>(*E)->ErrorCode==
           DBISAM_RECLOCKFAILED)
          {
          Action=uaRetry;
          }
       }
 }
```

Of course, there are many responses that can be made in this event handler depending upon the actual error code and any input that the user may be able to provide. The TDBISAMRecord object passed in contains both the current values and the old values of the record being applied, which allows you to prompt the user for an answer to a possible issue with a key violation, locking issue, or a record being modified by another user since it was last cached. In some cases, like duplicate key violations, it is possible to modify the current values so that the record can still be inserted, updated, or deleted.

Filters, Ranges, and Master-Detail Links

Most of the operations that can be performed on a TDBISAMTable or TDBISAMQuery component behave the same regardless of whether cached updates are active or not. This includes the following operations:

Navigating Tables and Query Result Sets
Searching and Sorting Tables and Query Result Sets
Updating Tables and Query Result Sets

However, certain states of the table or query result set are not carried over to the cached updates temporary table. These include:

Filters
Ranges
Master-Detail Links

All of these states are reset for the cached updates temporary table. You may apply new filters, ranges, and/or master-detail links on the cached updates temporary table if you wish, but they will not apply to the base table nor will they affect the base table's state with respect to filters, ranges, or master-detail links. After the cached updates are applied or cancelled, all of these states are set back to what they were prior to the cached updates being active.

Refreshing During Cached Updates

If you call the TDBISAMTable or TDBISAMQuery Refresh method while cached updates are active, then the current contents of the cached updates temporary table will be discarded and replaced with the latest data from the base table. Cached updates will remain in effect after the Refresh is complete.

Chapter 3
Advanced Topics

3.1 Locking and Concurrency

Introduction

DBISAM manages most locking and concurrency issues without requiring any action on the part of the developer. The following information details the steps that DBISAM takes internally in order to maximize concurrency while still resolving conflicts for shared resources using locking.

How DBISAM Performs Locking

All locks in DBISAM are performed using calls to the operating system. If using a local session accessing DBISAM tables on a network file server, these calls are then routed by the operating system to the file server's operating system, which could be Windows, Linux, etc. The benefit of this approach is that dangling locks left from an improper shutdown can be cleaned up by the operating system rather quickly.

DBISAM takes advantage of the fact that both Windows and the Linux operating systems allow an application to lock portions of a file beyond the actual size of the file. This process is known as virtual byte offset locking. DBISAM restricts the size of any physical data, index, or BLOB file that is part of a table to 128,000,000,000 bytes by default, or a little under 128 gigabytes. DBISAM does this so it can reserve the space available between the 128 gigabyte mark and the 128,000,000,000 byte mark for record and semaphore locks in the table. For table locks DBISAM uses a special hidden file called "dbisam.lck" (by default) that it automatically creates in the database directory where the tables are stored. This file is only used for keeping a list of the tables in the database and for placing virtual byte offset locks for table read, write, and transaction locks. Using this one file for table locks allows DBISAM to perform transaction locking without encountering deadlocks, which was an issue in past versions of DBISAM. The default lock file name "dbisam.lck" can be modified to any file name desired by modifying the TDBISAMEngine LockFileName property.

> **Note**
> If the lock file does not exist and cannot be created due to issues with security permissions, then the database will be treated as read-only and you will not be able to modify any tables in the database.

Record Locking Protocols

DBISAM offers two types of record locking protocols, pessimistic (default) and optimistic locking. The record locking protocol is configurable via the TDBISAMSession LockProtocol property.

Locking Model	Description

Pessimistic	The pessimistic record locking model specifies that a record should be locked when the record is retrieved for editing, which is during a call to the TDBISAMTable or TDBISAMQuery Edit method or during the record retrieval in an UPDATE SQL statement.
Optimistic	The optimistic locking model specifies that a record should be locked when any record modifications are posted to the table, which is during a call to the TDBISAMTable or TDBISAMQuery Post method or during the record modification in an UPDATE SQL statement. Using an optimistic record locking model for remote sessions removes the possibility that dangling record locks will be left on the database server if the application is terminated unexpectedly.

The two record locking protocols can safely and reliably be used among multiple sessions on the same set of tables, although it is not recommended due to the potential for confusion for the developer and user of the application.

User or Developer-Controlled Locks

There are three types of user or developer-controlled locks in DBISAM:

Record Locks
Table Locks
Semaphore Locks

Record locks are initiated by the user or developer when a record is appended, edited, or deleted. Table locks and semaphore locks, on the other hand, must be specifically set by the developer.

Record Locks

Record locks are used to enforce DBISAM's pessimistic or optimistic record locking protocols and prevent the same or multiple sessions from editing or posting modifications to the same record at the same time. Record locks block other record or table lock attempts, but do not block any reads of the locked records. The following details what happens in the various scenarios that use record locks:

Action	Description

Appending	When adding a record using the Append or Insert method of the TDBISAMTable or TDBISAMQuery component, no record locks are acquired until the record is posted using the Post method of the TDBISAMTable or TDBISAMQuery component. During the posting of a new record, a record lock is implicity acquired by DBISAM on the next available physical record. This record lock will fail only if the entire table is already locked by the same session or a different session. If the record lock fails, then an EDBISAMEngineError exception will be raised. The error code that is given when a record lock fails is 10258 and is defined as DBISAM_RECLOCKFAILED in the dbisamcn unit (Delphi) or dbisamcn header file (C++).
Editing	When editing a record using the Edit method of the TDBISAMTable or TDBISAMQuery component, a record lock is implicity acquired by DBISAM if the record locking protocol for the session is set to pessimistic (see above). This record lock will fail if the record or entire table is already locked by the same session or a different session. If the record lock fails, then an EDBISAMEngineError exception will be raised. The error code that is given when a record lock fails is 10258 and is defined as DBISAM_RECLOCKFAILED in the dbisamcn unit (Delphi) or dbisamcn header file (C++). If the locking protocol for the session is set to optmistic then the Edit method will not attempt to implicitly acquire a record lock, but will instead wait until the Post method is called to implicitly acquire the record lock. This means that another session is capable of editing the record and posting the changes to the record before the Post method is called. If this occurs, then an EDBISAMEngineError exception will be raised. The error code that is given when a call to the Post method fails because the record has been altered is 8708 and is defined as DBISAM_KEYORRECDELETED in the dbisamcn unit (Delphi) or dbisamcn header file (C++). In such a case you must discard the edited record by calling the Cancel method, call the Refresh method to refresh the record, and begin again with a fresh copy of the record using the Edit method.
Deleting	When deleting a record using the Delete method of the TDBISAMTable or TDBISAMQuery component, a record lock is implicity acquired by DBISAM. This record lock will fail if the record or entire table is already locked by the same session or a different session. If the record lock fails, then an EDBISAMEngineError exception will be raised. The error code that is given when a record lock fails is 10258 and is defined as DBISAM_RECLOCKFAILED in the dbisamcn unit (Delphi) or dbisamcn header file (C++). If another session edits the record and posts the changes to the record before the Delete method is called, an EDBISAMEngineError exception will be raised. The error code that is given when a call to the Delete method fails because the record has been altered is 8708 and is defined as DBISAM_KEYORRECDELETED in the dbisamcn unit (Delphi) or dbisamcn header file (C++). In such a case you must call the Refresh method to refresh the record and begin again with a fresh copy of the record using the Delete method.

Table Locks

Table locks are used to allow the developer to prevent any other sessions from adding, editing, or deleting any records or placing any record or table locks on a given table. Table locks block other record or table lock attempts, but do not block any reads of the locked table. A table lock is equivalent to locking all of the records in a table, including any records that may be added in the future. Table locks are always pessimistic and are not affected by the record locking protocol in use for record locks.

The TDBISAMTable LockTable method is used to acquire a table lock. If the table lock fails, then an EDBISAMEngineError exception will be raised. The error code that is given when a table lock fails is 10241 and is defined as DBISAM_LOCKED in the dbisamcn unit (Delphi) or dbisamcn header file (C++). The TDBISAMTable UnlockTable method is used to remove a table lock. The following is an example of using the LockTable and UnlockTable methods of the TDBISAMTable component:

```
{
   MyTable->LockTable;
   try
      {
      // Perform some updates to the table
      }
   __finally
      {
      MyTable->UnlockTable();
      }
}
```

Locking all of the records in a table using the LockTable method is useful for ensuring that no other users or processes make changes to a given table while a batch process is executing.

Semaphore Locks

Semaphore locks are used to provide access serialization in specific user-defined application functionality such as batch updates or system configuration updates and are not required in the normal operation of DBISAM. Semaphore locks can be placed in what are simply referred to as slots, and these slots are numbered from 1 to 1024. Semaphore locks only block other semaphore lock attempts for the same slot.

> **Note**
> Semaphore locks are table-based, with a different set of semaphore slots per table.

The TDBISAMTable LockSemaphore method is used to place a semaphore lock. If the semaphore lock fails, then the result of the LockSemaphore method will be False. The TDBISAMTable UnlockSemaphore method is used to remove a semaphore lock. The following is an example of using the LockSemaphore and UnlockSemaphore methods of the TDBISAMTable component:

```
{
   if (MyTable->LockSemaphore(1))
      {
      try
         {
```

```
          // Perform a batch process
          }
       __finally
          {
          MyTable->UnlockSemaphore(1);
          }
       }
}
```

Lock Retry Count and Wait Time

The number of record and table lock retries and the amount of time between each retry can be controlled using the TDBISAMSession LockRetryCount and LockWaitTime properties. In a busy multi-user application it may be necessary to increase these values in order to relieve lock contention and provide for smoother concurrency between multiple users. However, in most cases the default values should work just fine.

Internal Locks Used by the Engine

There are three types of internal locks in DBISAM:

Table Read Locks
Table Write Locks
Database Transaction Locks

Table tead locks are used by DBISAM to allow reads by multiple sessions while blocking any table write locks. Table read locks do not block other table read lock attempts. Table write locks, on the other hand, are used to serialize writes to a given table and therefore block any table read lock attempts or table write lock attempts.

Table Read Locks

Table read locks allow DBISAM to accurately treat reads on internal table structures such as the indexes or BLOB fields as atomic, or a single unit of work. Table read locks ensure that no other session writes to the table by blocking any table write locks. Table read locks are the most widely-used locks in DBISAM and are the cornerstone of correct multi-user operation. They especially play a large role in change detection. Please see the Change Detection topic for more information.

Table read locks are also acquired during table scans for un-optimized filter or query conditions. You can control the maximum number of table read locks acquired during a table scan via the TDBISAMEngine TableMaxReadLockCount property. Please see the Filter Optimization topic for more information on how filter conditions are optimized, and the SQL Optimizations topic for more information on optimizing SQL query conditions.

Table Write Locks

Table write locks allow DBISAM to accurately treat writes on internal table structures such as the indexes or BLOB fields as atomic, or a single unit of work. Table write locks ensure that no other session reads from or writes to the table by blocking any table read lock or write locks.

Database Transaction Locks

Database transaction locks allow DBISAM to treat multi-table updates within a transaction as atomic, or a single unit of work. Database transaction locks ensure that no other session writes to the database by

blocking any table write locks while the transaction is in effect. Table read locks are allowed, however, and other sessions can read the data from tables and acquire record and table locks. During the commit of a transaction, the database transaction lock is escalated so that table read locks are also blocked while the transaction is written to the database.

3.2 Buffering and Caching

Introduction

DBISAM uses caching and buffering algorithms internally to ensure that data is cached for as long as possible and is accessible in the fastest possible manner when needed to perform an operation. The following information details these internal processes.

Buffer Replacement Policy

Any buffer maintained within DBISAM is replaced using a LRU, or least-recently-used, algorithm. For example, if the cache is full when reading a record, DBISAM will discard the least-recently-used record buffer in order to make room for the new record buffer. The "age" of a given buffer is determined by the access patterns at the time. Every time a buffer is accessed it is moved so it is the first buffer in the list of available buffers. This would make it the "youngest" buffer present in the list of available buffers, and all other buffers would be moved down the list. As a particular buffer moves down the list it becomes "older" and will be more likely to be removed and discarded from the list of available buffers.

Read Ahead Buffering

DBISAM performs intelligent read-ahead when reading records and BLOB blocks. For read-ahead on records, this intelligence is gathered from information in the active index for a given table and allows DBISAM to determine how records physically align with one another on disk. Performing read-ahead in this manner can reduce the number of I/O calls that DBISAM has to make to the operating system and can significantly speed up sequential read operations such as those found in SQL queries and other bulk operations.

Block Writes

When DBISAM writes data to disk it aligns the data according to its physical placement on disk and attempts to write all of the needed data in the fewest number of I/O calls that is possible. This reduces the number of I/O calls and can make commit operations for transactions extremely quick, especially for bulk appends of records within a transaction.

OS Buffering

In addition to the buffering provided by DBISAM, additional buffering may be provided by the operating system in use. When DBISAM writes data using operating system calls, there is no guarantee that the data will be immediately written to disk. On the contrary, it may be several seconds or minutes until the operating system lazily flushes the data to disk. This has implications in terms of data corruption if the workstation is improperly shut down after updates have taken place in DBISAM. You can get around this by using the TDBISAMSession ForceBufferFlush property or by using the TDBISAMTable or TDBISAMQuery FlushBuffers method. The most desirable way to ensure that data is flushed to disk at the operating system level is the FlushBuffers method since the ForceBufferFlush property is very disk-intensive and may cause write performance to drop below an acceptable level. The FlushBuffers method, on the other hand, can be used in critical places in an application to ensure that data is flushed to disk in a timely fashion without necessarily sacrificing performance.

Modifying the Amount of Buffering

DBISAM enables you to modify the amount of memory used for buffering each table's record, index, and

BLOB field data. Please see the Customizing the Engine topic for more information.

3.3 Change Detection

Introduction

DBISAM automatically uses the proper change detection when dealing with updates to tables. However, there are two different types of change detection policies that can be used when dealing with reading data from tables:

Strict Change Detection
Lazy Change Detection

The choice of which policy to use is up to the developer and his/her needs and can be controlled via the TDBISAMSession StrictChangeDetection property. Also, any time DBISAM checks for changes in a given table it acquires a read lock on the table so as to ensure that no other changes occur while DBISAM performs the actual checks. Please see the Locking and Concurrency topic for more information.

Strict Change Detection

Strict change detection uses a "brute-force" method of determining whether the data has been changed by session during the process of reading data from a table. What this means is that every operation that requires reading of data from a table such as moving between records, filtering, setting ranges, searching, etc. will cause DBISAM to check for changes before the operation is executed. If DBISAM finds that the data in the table has changed, it will dump the contents of its local cache and refresh it using the latest data from the table. This can have some very significant performance implications, especially when the table resides on a network file server, so you should use this policy only when it is absolutely necessary that the data being read is always up-to-date at the time of the operation. It also tends to completely defeat the local caching done by DBISAM if there are a lot of updates taking place concurrently on the same tables. Please see the Buffering and Caching topic for more information.

Note
Strict change detection does not guarantee that the data you currently see is the latest data, only that the next time you perform a read operation you will see the latest data. DBISAM does not perform polling or background operations to constantly check for changes, only when it is instructed to perform a read operation.

Lazy Change Detection

Lazy change detection is the default change detection policy and is the most desirable in terms of efficiency and performance. Lazy change detection works by only checking for changes by other sessions when DBISAM cannot find the desired data locally in its cache and must physically read the data from the table. If changes are found, DBISAM will dump its cache and retry the read operation that it was in the process of executing when it found that it needed more data from the table. Because of the fact that DBISAM can cache a fairly large amount of data for each table open within a session, this policy tends to be very efficient and will provide the best performance overall. However, it does leave the job of refreshing data up to the developer so please take this into account when developing an application using this change detection policy.

Note
The amount of memory used for buffering tables can affect how often DBISAM detects changes within tables using lazy change detection, and DBISAM allows you to change these settings. Please see the Customizing the Engine topic for more information.

Updates and Change Detection

DBISAM always uses a strict change detection policy when performing updates. This means that anytime you append, edit, or delete a record DBISAM will automatically make sure that it's local cache contains the most up-to-date data before performing the actual update operation. In addition to this, DBISAM also performs a record buffer comparison when editing or deleting records to make sure that the record that is now present in it's cache is consistent with the record that was intended to be edited or deleted before the operation was initiated (i.e. it's what the user sees). If the record is not the same due to a change or deletion by another user or session, DBISAM will trigger the error DBISAM_KEYORRECDELETED indicating that the record has been changed or deleted by another user and the operation will be aborted. This record buffer comparison also includes the comparison of BLOB "signatures" in the record buffer so it is safe when determining if BLOB fields have changed also.

3.4 Index Compression

Introduction

DBISAM provides different ways of specifying how indexes should be compressed when creating or altering the structure of tables, as well as adding new indexes to a table. Please see the Creating and Altering Tables and Adding and Deleting Indexes from a Table topics for more information. The following information details the different types of index compression and how they should be used.

Types of Compression

The four different types of index compression available are:

Type	Description
No Compression	In most cases it is not very useful to specify no compression at all since almost every type of index can benefit from some type of compression. The exception to this would be primary or unique secondary indexes that are comprised of only one SmallInt, Word, or very short (< 4 characters) String type of field.
Duplicate-Byte Compression	Duplicate-byte compression works by comparing a given index key to its prior index key on the same index page and removing any duplicate bytes (working from the beginning of the index key to the end).
Trailing-Byte Compression	Trailing-byte compression works by removing any trailing blank or null bytes from a given index key (working from the end of the index key to the beginning).
Full Compression	Full compression works by combining both duplicate-byte compression with trailing-byte compression at the same time.

Compression Recommendations

If you are using only non-String fields in an index key and the index is not unique (or primary), then the highest compression level you should specify is duplicate-byte compression. You should not use trailing-byte compression in such a case at all since it will most likely provide very little benefit for most scalar data types (Integer, SmallInt, Word, Boolean, etc.).

If you're using a String field at the end of an index key and the index is not unique (or primary), then you should specify full compression, since this will not only remove duplicate bytes from the beginning of the index key it will also remove any trailing blanks or nulls from the end of the index key. This is especially true with indexes with large index key sizes. However - if the String field at the end of the index key is always filled entirely (such as may be the case with an ID field or something similar) then you should only use duplicate-byte compression for the index. Trailing-byte compression is most effective with large String fields that have a high likelihood of not being filled to capacity very often, such as is the case with an address or company name field.

If you're using only a String field in an index and the index is unique (or primary), you should verify whether the index will be smaller with just the trailing-byte compression specified. The amount of possible compression for the full compression option in this case is limited with unique indexes because there will be a smaller likelihood of duplicate bytes at the beginning of the index keys. It really is a factor of the data

values in the table, so you have to experiment a little.

If you're using only a non-String field in an index and the index is unique (or primary), you should verify whether the index will be smaller with no compression specified. The amount of possible compression for the duplicate-byte compression option in this case is limited with unique indexes because there will be a smaller likelihood of duplicate bytes at the beginning of the index keys. This is also a factor of the data values in the table, so again you have to experiment a little.

3.5 Filter Optimization

Introduction

DBISAM's filter optimizations rely on the use of available indexes and bitmaps in order to facilitate the quick retrieval and manipulation of sets of records that satisy all, or a portion of, a set of filter constraints.

Setting the Filter Expression

When an expression filter is set on a table in DBISAM using the TDBISAMTable or TDBISAMQuery Filter property, the following steps take place:

1) The filter expression is parsed and a set of token objects is created for each token in the expression.

2) The set of token objects is then examined for proper syntax and any errors in the filter expression are reported at this time.

3) The set of token objects is then examined again in order to determine the optimization level and make it available to the developer for examination via the TDBISAMTable or TDBISAMQuery FilterOptimizeLevel property. This process looks at the available indexes for each filter condition and uses this information to determine how optimized the filter expression is.

4) Once the filter is activated via the TDBISAMTable or TDBISAMQuery Filtered property, the optimization and filtering processes are performed.

How DBISAM Selects Indexes for Optimization

The first step in the optimization process is determining which indexes are available that can be used to speed up the filtering process. The rules for this index selection are as follows:

1) DBISAM only uses the first field of any given index for optimization. This means that if you have an index containing the fields LastName and FirstName, then DBISAM can only use this index for optimizing any filter conditions that refer to the LastName field.

2) DBISAM can use both ascending and descending indexes for optimization.

3) DBISAM will only use case-sensitive indexes for optimizing any filter conditions on string fields unless the foCaseInsensitive option is used with the TDBISAMTable or TDBISAMQuery FilterOptions property. You may also use the UPPER() or LOWER() functions on a column name to force DBISAM to use a case-insensitive index for optimizing the filter condition. Filter conditions on non-string fields such as integer or boolean fields can always use any index that contains the same field, regardless of the index's case-insensitivity setting.

4) DBISAM can mix and match the optimization of filter conditions so that it is possible to have one condition be optimized and the other not. This is known as a partially-optimized filter.

How DBISAM Builds the Filter Results

Once an index is selected for optimizing a given condition of the filter expression, a range is set on the index in order to limit the index keys to those that match the current filter condition being optimized. The index keys that satisfy the filter condition are then scanned, and during the scan a bitmap is built in physical record number order. A bit is turned on if the physical record satisfies the condition, and a bit is

turned off if it doesn't. This method of using bitmaps works well because it can represent sets of data with minimal memory consumption. Also, DBISAM is able to quickly determine how many records are in the set (how many bits are turned on), and it can easily AND, OR, and NOT bitmaps together to fulfill boolean logic between multiple filter conditions. Finally, because the bitmap is in physical record order, accessing the records using a bitmap is very direct since DBISAM uses fixed-length records with directly-addressable offsets in the physical table format.

Further Optimizations Provided by DBISAM

In addition to just using indexes to speed up the filtering process, DBISAM also provides a few other optimizations that can greatly increase a given filter's performance. When building a bitmap for a given optimized condition, DBISAM can take advantage of statistics that are kept in DBISAM indexes. These statistics accurately reflect the current make-up of the various values present in the index.

DBISAM looks at the optimization of the filter conditions, and when multiple conditions are joined by an AND operator, DBISAM ensures that the most optimized filter condition is executed first. For example, consider a table of 25,000 records with the following structure:

```
Customer table

Field       Data Type      Index
-------------------------------------------------
ID          Integer        Primary Index
Name        String[30]
State       String[2]      Secondary, case-sensitive,
                           non-unique, ascending, index
TotalOrders BCD[2]
```

And consider the following filter:

```
(TotalOrders > 10000) and (State='CA')
```

As you can see, the TotalOrders condition cannot be optimized since no indexes exist that would allow for optimization, whereas the State condition can be optimized. If only 200 records in the table have a State field that contains 'CA', then processing the filter in the order indicated by the expression would be very inefficient, since the following steps would take place:

1) All 25,000 physical records would be read and evaluated to build a bitmap for the (TotalOrders > 10000) condition.

2) The resultant bitmap from the previous step would be ANDed together with a bitmap built using the optimized index scan for the State condition.

DBISAM uses a much better approach because it knows that:

1) The TotalOrders condition is not optimized

2) The State condition is optimized

3) Both conditions are joined using the AND operator

it is able to reverse the filter conditions in the WHERE clause and execute the index scan for the 200 records that satisfy the State condition first, and then proceed to only read the 200 records from disk in order to evaluate the TotalOrders condition. DBISAM has just saved a tremendous amount of I/O by simply reversing the filter conditions.

> **Note**
> This optimization only works with filter conditions that are joined by the AND operator. If the above two conditions were joined using the OR operator, then DBISAM would simply read all 25,000 records and evaluate the entire filter expression for each record.

In the case of a completely un-optimized filter, DBISAM's read-ahead buffering can help tremendously in reducing network traffic and providing the most efficient reads with the least amount of I/O calls to the operating system. DBISAM will read up to 32 kilobytes of contiguous records on disk in the course of processing an un-optimized filter.

DBISAM can also optimize for the UPPER() and LOWER() functions by using any case-insensitive indexes in the table to optimize the filter condition. Take the following table for example:

```
Customer table

Field       Data Type      Index
-------------------------------------------------
ID          Integer        Primary Index
Name        String[30]
State       String[2]      Secondary, case-insensitive,
                           non-unique, ascending, index
```

And consider the following filter:

```
(UPPER(State)='CA')
```

In this filter, DBISAM will be able to select and use the case-insensitive index on the State field, and this is caused by the presence of the UPPER() function around the field name.

Optimization Levels

DBISAM determines the level of optimization for a filter using the following rules:

```
Optimized Condition = Fully-Optimized filter

Un-Optimized Condition = Un-Optimized filter

Optimized Condition AND Optimized Condition = Fully-
Optimized filter

Optimized Condition AND Un-Optimized Condition = Partially-
Optimized filter

Un-Optimized Condition AND Optimized Condition = Partially-
```

```
Optimized filter

Un-Optimized Condition AND Un-Optimized Condition = Un-
Optimized filter

Optimized Condition OR Optimized Condition = Fully-
Optimized filter

Optimized Condition OR Un-Optimized Condition = Un-
Optimized filter

Un-Optimized Condition OR Optimized Condition = Un-
Optimized filter

Un-Optimized Condition OR Un-Optimized Condition = Un-
Optimized filter
```

Note
The unary NOT operator causes any expression to become partially optimized. This is due to the fact that DBISAM must scan for, and remove, deleted records from the current records bitmap once it has taken the bitmap and performed the NOT operation on the bits.

DBISAM Limitations

DBISAM does not optimize multiple filter conditions joined by an AND operator) by mapping them to a compound index that may be available. To illustrate this point, consider a table with the following structure:

```
Employee

Field       Data Type      Index
------------------------------------------------------------------------
LastName    String[30]     Primary Index  (both fields are part of the
FirstName   String[20]     Primary Index   primary index)
```

And consider the following filter:

```
(LastName='Smith') and (FirstName='John')
```

Logically you would assume that DBISAM can use the one primary index in order to optimize the entire filter. Unfortunately this is not the case, and instead DBISAM will only use the primary index for optimizing the LastName condition and resort to reading records in order to evaluate the FirstName condition.

3.6 Multi-Threaded Applications

Introduction

DBISAM is internally structured to be thread-safe and usable within a multi-threaded application provided that you follow the rules that are outlined below.

Unique Sessions

DBISAM requires that you use a unique TDBISAMSession component for every thread that must perform any database access at all. Each of these TDBISAMSession components must also contain a SessionName property that is unique among all TDBISAMSession components in the application. Doing this allows DBISAM to treat each thread as a separate and distinct "user" and will isolate transactions and other internal structures accordingly. You may use the AutoSessionName property of the TDBISAMSession component to allow DBISAM to automatically name each session so that is unique or you may use code similar to the following:

```
   int LastSessionValue;
   TRTLCriticalSection SessionNameSection;

// Assume that the following code is being executed
//  within a thread

bool __fastcall UpdateAccounts();
{
   bool TempResult=false;
   TDBISAMSession *LocalSession=GetNewSession;
   try
      {
      TDBISAMDatabase *LocalDatabase=new TDBISAMDatabase(NULL);
      try
         {
         // Be sure to assign the same session name
         // as the TDBISAMSession component
         LocalDatabase->SessionName=LocalSession->SessionName;
         LocalDatabase->DatabaseName="Accounts";
         LocalDatabase->Directory="g:\\accountdb";
         LocalDatabase->Connected=true;
         TDBISAMQuery *LocalQuery=new TDBISAMQuery(NULL);
         try
            {
            // Be sure to assign the same session and
            // database name as the TDBISAMDatabase
            // component
            LocalQuery->SessionName=LocalSession->SessionName;
            LocalQuery->DatabaseName=LocalDatabase->DatabaseName;
            LocalQuery->SQL->Clear();
            LocalQuery->SQL->Add("UPDATE accounts SET PastDue=True");
            LocalQuery->SQL->Add("WHERE DueDate < CURRENT_DATE"));
            LocalQuery->Prepare;
            try
               {
               // Start the transaction and execute the query
```

```
               LocalDatabase->StartTransaction();
               try
                  {
                  LocalQuery->ExecSQL();
                  LocalDatabase->Commit();
                  TempResult=true;
               catch
                  {
                  LocalDatabase->Rollback();
                  }
            __finally
               {
               LocalQuery->UnPrepare();
               }
            }
         __finally
            {
            delete LocalQuery;
            }
         }
      __finally
         {
         delete LocalDatabase;
         }
      }
   __finally
      {
      delete LocalSession;
      }
   return TempResult;
end;

TDBISAMSession* __fastcall GetNewSession();
{
   TDBISAMSession *TempResult=NULL;
   EnterCriticalSection(SessionNameSection);
   try
      {
      LastSessionValue=(LastSessionValue+1);
      TDBISAMSession *TempResult=new TDBISAMSession(NULL);
      TempResult->SessionName="AccountSession"+
                        IntToStr(LastSessionValue);
      }
   __finally
      {
      LeaveCriticalSection(SessionNameSection);
      }
   return TempResult;
}

{ initialization in application }
   LastSessionValue=0;
   InitializeCriticalSection(SessionNameSection);
{ finalization in application }
   DeleteCriticalSection(SessionNameSection);
```

The AutoSessionName property is, by default, set to False so you must specifically turn it on if you want this functionality. You may also use the thread ID of the currently thread to uniquely name a session since

the thread ID is guaranteed to be unique within the context of a process.

Unique Databases

Another requirement is that all TDBISAMDatabase components must also be unique and have their SessionName properties referring to the unique SessionName property of the TDBISAMSession component defined in the manner discussed above.

Unique Tables and Queries

The final requirement is that all TDBISAMTable and TDBISAMQuery components must also be unique and have their SessionName properties referring to the unique SessionName property of the TDBISAMSession component defined in the manner discussed above. Also, if a TDBISAMTable or TDBISAMQuery component refers to a TDBISAMDatabase component's DatabaseName property via its own DatabaseName property, then the TDBISAMDatabase referred to must be defined in the manner discussed above.

ISAPI Applications

ISAPI applications created using the Borland WebBroker components or a similar technology are implicitly multi-threaded. Because of this, you should ensure that your ISAPI application is thread-safe according to these rules for multi-threading when using DBISAM. Also, if you have simply dropped a TDBISAMSession component on the WebModule of a WebBroker ISAPI application, you must set its AutoSessionName property to True before dropping any other DBISAM components on the form so that DBISAM will automatically give the TDBISAMSession component a unique SessionName property and propogate this name to all of the other DBISAM components.

Further Considerations

There are some other things to keep in mind when writing a multi-threaded database application with DBISAM, especially if the activity will be heavy and there will be many threads actively running. Be prepared to handle any errors in a manner that allows the thread to terminate gracefully and properly free any TDBISAMSsion, TDBISAMDatabase, TDBISAMTable, or TDBISAMQuery components that it has created. Otherwise you may run into a situation where memory is being consumed at an alarming rate. Finally, writing multi-threaded applications, especially with database access, is not a task for the beginning developer so please be sure that you are well-versed in using threads and how they work before jumping into writing a multi-threaded application with DBISAM.

3.7 Full Text Indexing

Introduction

DBISAM provides the ability to index string and memo fields so that they may be quickly searched for a given word or words. This is known as full text indexing since it results in the indexing of every word in every column that is specified as part of the full text index for the table. This whole process is controlled by full text indexing parameters that are defined as part of the table structure when creating or altering the structure of tables, as well as events in the TDBISAMEngine component for customizing the full text indexing. Please see the Customizing the Engine for more information.

> **Note**
> Full text indexing and searching is always case-insensitive in DBISAM. This means that words are always compared without regard for case, however the include and space character full text indexing parameters are compared on an exact character basis when parsing the text to be indexed or searched.

Text Indexing Parameters

The three parameters that control the full text indexing behavior for a given table are:

Parameter	Description
Stop Words List	The stop words list is a list of words that are to be excluded from the full text index. These words are usually very common words and excluding them from the full text index can result in tremendous space savings for the physical index. The default stop words for a table are as follows: A AN AND BE FOR HOW IN IS IT OF ON OR THAT THE THIS TO WAS WHAT WHEN WHICH WHY WILL

	The stop words list is always case-insensitive, as is the full text indexing in general.
Space Characters	The space characters specify which characters in the ANSI character set are to be used for word separator characters. These characters usually consist of any character below the ordinal value of 33 and other separators such as backslashes (\)and commas (,). The default space characters for a table are as follows: Characters 1 through 32 *+,-./:;<=>\
Include Characters	The include characters specify which characters in the ANSI character set are to be included in the words that are finally used for the full text index. These characters usually consist of all alphanumeric characters as well as all high character values in the ANSI character set that are used by non-English languages for accented characters and other diacritically-marked characters. The default include characters for a table are as follows: 0123456789 ABCDEFGHIJKLMNOPQRSTUVWXYZ_ abcdefghijklmnopqrstuvwxyz €‚ƒ„…†‡ˆ‰Š‹ŒŽ‘’“”•–—˜™ š›œžŸ ¡¢£¤¥¦§¨©ª«¬®¯°±²³ ´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍ ÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞßàáâãäåæç èéêëìíîïðñòóôõö÷øùúûüýþ

Note
You must alter the structure of a table in order to change any of these parameters.

Performing a Text Search

DBISAM includes the filter and SQL TEXTSEARCH function in order to take advantage of the full text index and also as a general-purpose, brute-force, word search function when searching on string or memo columns that are not part of the full text index. The TEXTSEARCH function accepts a list of words in a search string constant and a column name as its two parameters. The following is an example of using the TEXTSEARCH function in an expression filter:

```
{
   MyTable->Filter="TEXTSEARCH("+QuotedStr("DATABASE QUERY SPEED")+
                   " IN TextBody)";
   MyTable->Filtered=true;
}
```

In the above example, if the TextBody column is included as part of the indexed fields that make up the full text index then the filter will execute very quickly. If the column is not part of the full text index, then the filter will be forced to resort to a brute-force scan of the TextBody column for every record in the

table. To further explain how the text searching works, let's break down the previous process (assuming an optimized text search):

1) DBISAM parses the search string constant "DATABASE QUERY SPEED" into three words (DATABASE, QUERY, and SPEED) using the space characters and include characters specified for the table, which by default would allow for the space character ( ) as the word separator in this case. If there happened to be a backslash in the search string such as "DATABASE C:\TEMP" then the search string would be parsed into three words "DATABASE C TEMP". This is because the default full text indexing space characters include the colon (:) and the backslash (\).

2) DBISAM takes the list of words created from the text indexing parameters and performs a case-insensitive search of the stop words for the table to see if any of the words exist in the stop words. If one or more does, then they are ignored when performing the actual search.

3) Finally, DBISAM searches the full text index and builds a bitmap for each word indicating which records satisfy the search for that particular word. These bitmaps are ANDed together and the resultant bitmap is used for filtering the records in the table. This process is very similar to a what happends with a normal optimized filter expression or SQL WHERE clause.

Note
DBISAM only executes an AND search for the multiple words in the search string. If you want to execute an OR search for multiple words you should split up the operation into multiple TEXTSEARCH calls as the following example illustrates:

```
{
   MyTable->Filter="TEXTSEARCH("+QuotedStr("DATABASE")+
                   " IN TextBody) OR "+
                   "TEXTSEARCH("+QuotedStr("QUERY")+
                   " IN TextBody) OR "+
                   "TEXTSEARCH("+QuotedStr("SPEED")+
                   " IN TextBody)";
   MyTable->Filtered=true;
}
```

This will give the desired results of returning all records where either DATABASE, QUERY, or SPEED are found in the TextBody column.

You can also specify partial-word searches using the asterisk as a trailing wildcard character. The following is an example of using the TEXTSEARCH function in an expression filter with a partial-word search string:

```
{
   MyTable->Filter="TEXTSEARCH("+QuotedStr("DATA*")+
                   " IN TextBody)";
   MyTable->Filtered=true;
}
```

Note
You can mix and match partial words and whole words in the same search string.

If you wish to find out what words were searched on in a TEXTSEARCH function or just want to test various text indexing parameters, you can do so using the TDBISAMEngine BuildWordList method.

Retrieving the Number of Occurrences

DBISAM includes the filter and SQL TEXTOCCURS function in order to calculate the number of times a specific list of words in a given search string occurs in a string or memo column. This function is always a brute-force function and should not be used for the bulk of the filtering or searching but rather for ranking purposes or something similar after the search has been completed using the TEXTSEARCH function. The TEXTOCCURS function accepts a list of words in a search string constant and a column name as its two parameters. The following is an example of using the TEXTOCCURS function in an SQL SELECT statement:

```
SELECT GroupNo, No, TEXTOCCURS('DATABASE QUERY SPEED' IN TextBody) AS
      NumOccurs
FROM article
WHERE TEXTSEARCH('DATABASE QUERY SPEED' IN TextBody)
ORDER BY 3 DESC
```

As you can see, the TEXTOCCURS function is being used to provide ranking of the results by the number of times the search words occur in the TextBody column after the bulk of the search was already handled by the optimized TEXTSEARCH function.

3.8 Compression

Introduction

DBISAM uses the standard ZLib compression algorithm for compressing data such as BLOB fields and remote session requests and responses to and from a database server. The ZLib compression is contained within the zlibpas and zlibcomp units (Delphi) or zlibpas and zlibcomp header files (C++).

Copyright and Credits

The ZLib implementation in DBISAM was contributed by David Martin. The following are the citations and copyrights for both the code that was contributed as well as for the ZLib algorithm itself.

© Copyright 1995-98 Jean-loup Gailly & Mark Adler
© Copyright 1998-00 Jacques Nomssi Nzali
© Copyright 2000-2001 David O. Martin

These units build upon a pascal port of the ZLib compression routines by Jean-loup Gailly and Mark Adler. The original pascal port was performed by Jacques Nomssi Nzali as contained in PasZLib which is based on ZLib 1.1.2. There are some errors in that port which have been fixed in this version. Although most of the code in this unit is derivative, there are some important changes (bug fixes). Nevertheless, this code is released as freeware with the same permissions as granted by the preceding authors (Gailly, Adler, Nzali).

Replacing the Default Compression

You can replace the default compression implementation in DBISAM by using the events provided in the TDBISAMEngine component. Please see the Customizing the Engine topic for more information.

Note
If you replace the default compression, keep in mind that you must make sure to not mix and match different compression implementations with various applications or servers that access the same databases and tables. Doing so can cause some serious problems and the potential for losing data since one application or server will not be able to read or write data that was compressed using a different compression implementation.

3.9 Encryption

Introduction

DBISAM uses the Blowfish symmetric block cipher encryption algorithm along with the RSA Data Security, Inc. MD5 message-digest algorithm for encrypting tables and remote session requests and responses to and from a database server. Both of these algorithms are contained within the dbisamcr unit (Delphi) or dbisamcr header file (C++).

Copyright and Credits

Both the Blowfish and MD5 implementations in DBISAM were written by us. The following are the citations and copyrights for the Blowfish and MD5 algorithms.

Blowfish Algorithm © Copyright 1993 Bruce Schneier
MD5 Algorithm © Copyright 1991-1992, RSA Data Security, Inc.

DBISAM uses the MD5 message-digest algorithm to generate 128-bit MD5 hashes from plain-text passwords. These hashes are then used with the Blowfish 8-byte symmetric block cipher algorithm to encrypt the actual data.

Replacing the Default Encryption

You can replace the default block cipher encryption implementation in DBISAM by using the events provided in the TDBISAMEngine component. Please see the Customizing the Engine topic for more information. You can only replace the default encryption implementation with another 8-byte block cipher implementation.

Note
If you replace the default encryption, keep in mind that you must make sure to not mix and match different encryption implementations with various applications or servers that access the same databases and tables. Doing so can cause some serious problems and the potential for losing data since one application or server will not be able to read or write data that was encrypted using a different encryption implementation.

3.10 Recompiling the DBISAM Source Code

Introduction

In some cases you may want to change the DBISAM source code and recompile it to incorporate these changes into your application. However, you must first have purchased a version of DBISAM that includes source code to the engine in order to make changes to the source code.

Setting Search Paths

The first thing that you must do is make sure that any search paths, either global to DBISAM such as the Library Search Path or local to your project, are pointing to the directory or path where the DBISAM source code was installed. By default this directory or path is:

```
\<base directory>\<product>\<compiler> <n>\code\source
```

The <product> component of the path can be one of the following values:

Value	Description
DBISAM <type> Standard with Source	This indicates the standard version of DBISAM with source code
DBISAM <type> Client-Server with Source	This indicates the client-server version of DBISAM with source code

The <type> component of the product name will be either VCL or ODBC.

The <compiler> <n> component of the path indicates the development environment in use and the version number of the development environment. For example, for Delphi 6 this component would look like this:

```
Delphi 6
```

Setting Compiler Switches

The second thing that must be done is to make sure that the compiler switches that you are using are set properly for DBISAM. The build system used to compile DBISAM here at Elevate Software uses the dcc32.exe, dcc64.exe, and dcc command-line compilers provided with Delphi and C++ to compile DBISAM. The following switches are set during compilation and any other switches are assumed to be at their default state for the compiler:

```
$D-   Debug information off
$L-   Local symbols off
```

Note
These same switches are used to compile all DBISAM utilities and the DBISAM server project also.

A Word of Caution

Making changes to the DBISAM source code is not an easy task. A mistake in such changes could result in the loss of critical data and Elevate Software cannot be held responsible for any losses incurred from such changes. Occasionally our support staff may send a fix to a customer that owns the source code in order to facilitate a quicker turnaround on a bug report, but it is the responsibility of the customer to weigh the risks of implementing such a change with the possible problems that such a change could bring about. Elevate Software tries very hard to also assist any customers that do want to make changes to the DBISAM source code for custom purposes and will always make an attempt to guide the customer to a solution that fits their needs and is reliable in operation. In general, however, it is usually recommended that you use the generic customization facilities provided with DBISAM as opposed to making direct changes to the source code. Please see the Customizing the Engine topic for more information.

3.11 Replacement Memory Manager

Introduction

DBISAM uses a replacement memory manager for Delphi called FastMM with both the GUI and command-line database servers that ship with all DBISAM products. The FastMM memory manager is designed to be extremely efficient at allocating and de-allocating many small blocks of memory of similar sizes, which is primarily what DBISAM does during most operations. The memory manager is also designed to overcome issues with the default Delphi memory manager that cause it to perform very poorly with multi-threaded applications that run on servers with multiple processors due to its use of a single critical section for all access to the memory pool. This design becomes a major performance bottleneck in such situations.

You can download the FastMM memory manager from SourceForge here:

FastMM

Including the Memory Manager in an Application

In any cases that you wish to include the FastMM replacement memory manager in an application, there are only a couple of steps involved:

1) Open the Delphi project .dpr file.

2) Include the FastMM4 unit that is shipped with FastMM as the first unit in the project's USES clause.

> **Note**
> It is extremely important that the FastMM4 unit is the very first unit in the USES clause.

The following is an partial excerpt of the dbsrvr.dpr project file that is shipped with DBISAM for the GUI database server. It illustrates how to include the FastMM4 unit in the project's USES clause:

```
program dbsrvr;

uses

  {$I dbisamvr.inc}

  {$IFDEF MSWINDOWS}
  FastMM4,
  {$ENDIF}

  SysUtils,
```

Compatibility Issues

The FastMM replacement memory manager is for use only with Delphi and is not intended for use with C++.

Chapter 4
SQL Reference

4.1 Overview

Introduction

DBISAM does not support the complete ANSI SQL-92 specification. Rather, it supports a subset of the specification that includes the most widely used SQL statements for data manipulation and definition, in some cases with DBISAM extensions, as well as some SQL statements that are specific to DBISAM:

SQL Statement	Standard
SELECT	SQL-92 with DBISAM Extensions
INSERT	SQL-92 with DBISAM Extensions
UPDATE	SQL-92 with DBISAM Extensions
DELETE	SQL-92 with DBISAM Extensions
CREATE TABLE	SQL-92 with DBISAM Extensions
ALTER TABLE	SQL-92 with DBISAM Extensions
EMPTY TABLE	DBISAM-specific
OPTIMIZE TABLE	DBISAM-specific
EXPORT TABLE	DBISAM-specific
IMPORT TABLE	DBISAM-specific
VERIFY TABLE	DBISAM-specific
REPAIR TABLE	DBISAM-specific
UPGRADE TABLE	DBISAM-specific
DROP TABLE	SQL-92 with DBISAM Extensions
RENAME TABLE	DBISAM-specific
CREATE INDEX	SQL-92 with DBISAM Extensions
DROP INDEX	SQL-92 with DBISAM Extensions
START TRANSACTION	SQL-92 with DBISAM Extensions
COMMIT	SQL-92 with DBISAM Extensions
ROLLBACK	SQL-92 with DBISAM Extensions

4.2 Naming Conventions

Introduction

DBISAM requires that certain naming conventions be adhered to when executing SQL. The following rules and naming conventions apply to all supported SQL statements in DBISAM.

Table Names

ANSI-standard SQL specifies that each table name must be a single word comprised of alphanumeric characters and the underscore symbol (_). However, DBISAM's SQL is enhanced to support multi-word table names by enclosing them in double quotes ("") or square brackets ([]):

```
SELECT *
FROM "Customer Data"
```

DBISAM's SQL also supports full file and path specifications in table references for SQL statements being executed within a local session. Table references with path or filename extensions must be enclosed in double quotes ("") or square brackets ([]). For example:

```
SELECT *
FROM "c:\sample\parts"
```

or

```
SELECT *
FROM "parts.dat"
```

Note
It is not recommended that you specifiy the .dat file name extension in SQL statements for two reasons:

1) First of all, it is possible for the developer to change the default table file extensions for data, index, and BLOB files from the defaults of ".dat", ".idx", and ".blb" to anything that is desired. Please see the DBISAM Architecture topic for more information.

2) Using file paths and extensions at all in SQL statements makes the SQL less portable to other database engines or servers.

DBISAM's SQL also supports database name specifications in table references for SQL statements being executed within a remote session. Table references with database must be enclosed in double quotes ("") or square brackets ([]). For example:

```
SELECT *
FROM "\Sample Data\parts"
```

Note
The database name used with remote sessions is not a directory name like it is with local sessions. Instead, it must be a logical database name that matches that of a database defined on the database server that you are accessing with the SQL statement.

To use an in-memory table in an SQL statement within both local and remote sessions, just prefix the table name with the special "Memory" database name:

```
SELECT *
FROM "\Memory\parts"
```

Please see the In-Memory Tables topic for more information.

Column Names

ANSI-standard SQL specifies that each column name be a single word comprised of alphanumeric characters and the underscore symbol (_). However, DBISAM's SQL is enhanced to support multi-word column names. Also, DBISAM's SQL supports multi-word column names and column names that duplicate SQL keywords as long as those column names are enclosed in double quotes ("") or square brackets ([]) or prefaced with an SQL table name or table correlation name. For example, the following column name consists of two words:

```
SELECT E."Emp Id"
FROM employee E
```

In the next example, the column name is the same as the SQL keyword DATE:

```
SELECT weblog.[date]
FROM weblog
```

String Constants

ANSI-standard SQL specifies that string constants be enclosed in single quotes ('), and DBISAM's SQL follows this convention. For example, the following string constant is used in an SQL SELECT WHERE clause:

```
SELECT *
FROM customer
WHERE Company='ABC Widgets'
```

Note
String constants can contain any character in the ANSI character set except for the non-printable characters below character 32 (space). For example, if you wish to embed a carriage-return and line feed in a string constant, you would need to use the following syntax:

```
UPDATE customer SET Notes='ABC Widgets'+
#13+#10+'Located in New York City'
```

The pound sign can be used with the ordinal value of any ANSI character in order to represent that single character as a constant.

To streamline the above, you can use the TDBISAMEngine QuotedSQLStr method to properly format and escape any embedded single quotes or non-printable characters in a string constant. Please see the Executing SQL Queries topic for more information.

Date, Time, TimeStamp, and Number Constants

DBISAM's SQL uses ANSI/ISO date and number formatting for all date, time, timestamp (date/time), and number constants, which is consistent with ANSI-standard SQL except for missing support for date and time interval constants, which are not supported in DBISAM's SQL currently. The formats are as follows:

Constant	Format
Dates	The date format is yyyy-mm-dd where yyyy is the year (4 digits required), mm is the month (leading zero optional), and the day (leading zero optional).
Times	The time format is hh:mm:ss.zzz am/pm where hh is the hour (leading zero optional), mm is the minutes (leading zero optional), ss is the seconds (leading zero optional), zzz is the milliseconds (leading zero optional), and the am/pm designation for times using the 12-hour clock. The seconds and milliseconds are optional when specifying a time, as is the am/pm designation. If the am/pm designation is omitted, the time is expected to be in 24-hour clock format.
Timestamps (date/time)	The timestamp format is a combination of the date format and the time format with a space in-between the two formats.
Numbers	All numbers are expected to use the period (.) as the decimal separator and no monetary symbols must be used. DBISAM's SQL does not support scientific notation in number constants currently.

All date, time, and timestamp constants must be enclosed in single quotes (') when specified in an SQL statement. For example:

```
SELECT *
FROM orders
WHERE (saledate <= '1998-01-23')
```

Boolean Constants

The boolean constants TRUE and FALSE can be used for specifying a True or False value. These constants are case-insensitive (True=TRUE). For example:

```
SELECT *
FROM transfers
WHERE (paid = TRUE) AND NOT (incomplete = FALSE)
```

Table Correlation Names

Compliant with ANSI-standard SQL, table correlation names can be used in DBISAM's SQL to explicitly associate a column with the table from which it is derived. This is especially useful when multiple columns of the same name appear in the same query, typically in multi-table queries. A table correlation name is defined by following the table reference in the SQL statement with a unique identifier. This identifier, or table correlation name, can then be used to prefix a column name. The base table name is the default implicit correlation name, irrespective of whether the table name is enclosed in double quotes ("") or square brackets ([]). The base table name is defined as the table name for the DBISAM table not including the full path or any file extensions. For example, the base table name for the physical table "c:\temp\customer.dat" is "customer" as show in this example:

```
SELECT *
FROM "c:\temp\customer.dat"
LEFT OUTER JOIN "c:\temp\orders.dat"
ON (customer.custno = orders.custno)
```

You may also use the physical file name for the table as a table correlation name, although it's not required nor recommended:

```
SELECT *
FROM "customer.dat"
LEFT OUTER JOIN "orders.dat"
ON ("customer.dat".custno = "orders.dat".custno)
```

And finally, you may use a distinctive token as a correlation name (and prefix all column references with the same correlation name):

```
SELECT *
FROM "customer" C
LEFT OUTER JOIN "orders" O
ON (C.custno = O.custno)
```

Column Correlation Names

You can use the AS keyword to assign a correlation name to a column or column expression within a

DBISAM SQL SELECT statement, which is compliant with ANSI-standard SQL. Column correlation names can be enclosed in double quotes ("") and can contain embedded spaces. The following example shows how to use the AS keyword to assign a column correlation name:

```
SELECT
customer.company AS "Company Name",
orders.orderno AS "Order #",
sum(items.qty) AS "Total Qty"
FROM customer LEFT OUTER JOIN orders ON customer.custno=orders.custno
LEFT OUTER JOIN items ON orders.orderno=items.orderno
WHERE customer.company LIKE '%Diver%'
GROUP BY 1,2
ORDER BY 1
```

You may also optionally exclude the AS keyword and simply specify the column correlation name directly after the column, as shown here:

```
SELECT
customer.company "Company Name",
orders.orderno "Order #",
sum(items.qty) "Total Qty"
FROM customer LEFT OUTER JOIN orders ON customer.custno=orders.custno
LEFT OUTER JOIN items ON orders.orderno=items.orderno
WHERE customer.company LIKE '%Diver%'
GROUP BY 1,2
ORDER BY 1
```

Embedded Comments

Per ANSI-standard SQL, comments, or remarks, can be embedded in SQL statements to add clarity or explanation. Text is designated as a comment and not treated as SQL by enclosing it within the beginning /* and ending */ comment symbols. The symbols and comments need not be on the same line:

```
/*
  This is a comment
*/
SELECT SUBSTRING(company FROM 1 FOR 4) AS abbrev
FROM customer
```

Comments can also be embedded within an SQL statement. This is useful when debugging an SQL statement, such as removing one clause for testing.

```
SELECT company
FROM customer
/* WHERE (state = 'TX') */
ORDER BY company
```

Reserved Words

Below is an alphabetical list of words reserved by DBISAM's SQL. Avoid using these reserved words for the names of metadata objects (tables, columns, and indexes). An exception occurs when reserved words are used as names for metadata objects. If a metadata object must have a reserved word as it name, prevent the error by enclosing the name in double-quotes ("") or square brackets ([]) or by prefixing the reference with the table name (in the case of a column name).

```
ABS
ACOS
ADD
ALL
ALLTRIM
ALTER
AND
AS
ASC
ASCENDING
ASIN
AT
ATAN
ATAN2
AUTOINC
AVG
BETWEEN
BINARY
BIT
BLOB
BLOCK
BOOL
BOOLEAN
BOTH
BY
BYTES
CAST
CEIL
CEILING
CHAR
CHARACTER
CHARCASE
CHARS
COALESCE
COLUMN
COLUMNS
COMMIT
COMPRESS
CONCAT
CONSTRAINT
COS
COT
COUNT
CREATE
CURRENT_DATE
CURRENT_GUID
CURRENT_TIME
CURRENT_TIMESTAMP
DAY
DAYOFWEEK
DAYOFYEAR
```

```
DAYSFROMMSECS
DECIMAL
DEFAULT
DEGREES
DELETE
DELIMITER
DESC
DESCENDING
DESCRIPTION
DISTINCT
DROP
DUPBYTE
ELSE
EMPTY
ENCRYPTED
ESCAPE
EXCEPT
EXISTS
EXP
EXPORT
EXTRACT
FALSE
FLOAT
FLOOR
FLUSH
FOR
FORCEINDEXREBUILD
FROM
FULL
GRAPHIC
GROUP
GUID
HAVING
HEADERS
HOUR
HOURSFROMMSECS
IDENT_CURRENT
IDENTITY
IF
IFNULL
IMPORT
IN
INCLUDE
INDEX
INNER
INSERT
INT
INTEGER
INTERSECT
INTERVAL
INTO
IS
JOIN
KEY
LARGEINT
LAST
LASTAUTOINC
LCASE
LEADING
LEFT
```

```
LENGTH
LIKE
LOCALE
LOG
LOG10
LONGVARBINARY
LONGVARCHAR
LOWER
LTRIM
MAJOR
MAX
MAXIMUM
MEMO
MIN
MINIMUM
MINOR
MINSFROMMSECS
MINUTE
MOD
MONEY
MONTH
MSECOND
MSECSFROMMSECS
NOBACKUP
NOCASE
NOCHANGE
NOJOINOPTIMIZE
NONE
NOT
NULL
NUMERIC
OCCURS
ON
OPTIMIZE
OR
ORDER
OUTER
PAGE
PI
POS
POSITION
POWER
PRIMARY
RADIANS
RAND
RANGE
REDEFINE
RENAME
REPAIR
REPEAT
REPLACE
RIGHT
ROLLBACK
ROUND
RTRIM
RUNSUM
SECOND
SECSFROMMSECS
SELECT
SET
```

```
SIGN
SIN
SIZE
SMALLINT
SPACE
SQRT
START
STDDEV
STOP
SUBSTRING
SUM
TABLE
TAN
TEXT
TEXTOCCURS
TEXTSEARCH
THEN
TIME
TIMESTAMP
TO
TOP
TRAILBYTE
TRAILING
TRANSACTION
TRIM
TRUE
TRUNC
TRUNCATE
UCASE
UNION
UNIQUE
UPDATE
UPGRADE
UPPER
USER
VALUES
VARBINARY
VARBYTES
VARCHAR
VERIFY
VERSION
WEEK
WHERE
WITH
WORD
WORDS
WORK
YEAR
YEARSFROMMSECS
```

The following are operators used in DBISAM's SQL. Avoid using these characters in the names of metadata objects:

```
|
+
-
*
```

```
/
<>
<
>
.
;
,
=
<=
>=
(
)
[
]
#
```

4.3 Unsupported SQL

The following ANSI-standard SQL-92 language elements are not used in DBISAM's SQL:

```
ALLOCATE CURSOR (Command)
ALLOCATE DESCRIPTOR (Command)
ALTER DOMAIN (Command)
CHECK (Constraint)
CLOSE (Command)
CONNECT (Command)
CONVERT (Function)
CORRESPONDING BY (Expression)
CREATE ASSERTION (Command)
CREATE CHARACTER SET (Command)
CREATE COLLATION (Command)
CREATE DOMAIN (Command)
CREATE SCHEMA (Command)
CREATE TRANSLATION (Command)
CREATE VIEW (Command)
CROSS JOIN (Relational operator)
DEALLOCATE DESCRIPTOR (Command)
DEALLOCATE PREPARE (Command)
DECLARE CURSOR (Command)
DECLARE LOCAL TEMPORARY TABLE (Command)
DESCRIBE (Command)
DISCONNECT (Command)
DROP ASSERTION (Command)
DROP CHARACTER SET (Command)
DROP COLLATION (Command)
DROP DOMAIN (Command)
DROP SCHEMA (Command)
DROP TRANSLATION (Command)
DROP VIEW (Command)
EXECUTE (Command)
EXECUTE IMMEDIATE (Command)
EXISTS (Predicate)
FETCH (Command)
FOREIGN KEY (Constraint)
GET DESCRIPTOR (Command)
GET DIAGNOSTICS (Command)
GRANT (Command)
MATCH (Predicate)
NATURAL (Relational operator)
NULLIF (Expression)
OPEN (Command)
OVERLAPS (Predicate)
PREPARE (Command)
REFERENCES (Constraint)
REVOKE (Command)
SET CATALOG (Command)
SET CONNECTION (Command)
SET CONSTRAINTS MODE (Command)
SET DESCRIPTOR (Command)
SET NAMES (Command)
SET SCHEMA (Command)
SET SESSION AUTHORIZATION (Command)
```

```
SET TIME ZONE (Command)
SET TRANSACTION (Command)
TRANSLATE (Function)
USING (Relational operator)
```

4.4 Optimizations

Introduction

DBISAM uses available indexes when optimizing SQL queries so that they execute in the least amount of time possible. In addition, joins are re-arranged to allow for the least number of joins as possible since joins tend to be fairly expensive in DBISAM.

Index Selection

DBISAM will use an available index to optimize any expression in the WHERE clause of an SQL SELECT, UPDATE, or DELETE statement. It will also use an available index to optimize any join expressions between multiple tables. This index selection is based on the following rules:

1) DBISAM only uses the first field of any given index for optimization. This means that if you have an index containing the fields LastName and FirstName, then DBISAM can only use the this index for optimizing any conditions that refer to the LastName field.

2) DBISAM can use both ascending and descending indexes for optimization.

3) DBISAM will only use case-sensitive indexes for optimizing any conditions on string fields unless the condition contains the UPPER() or LOWER() SQL function. In such a case DBISAM will only look for and use case-insensitive indexes for optimizing the condition. Conditions on non-string fields such as integer or boolean fields can always use any index that contains the same field, regardless of the index's case-insensitivity setting.

4) DBISAM can mix and match the optimization of conditions so that it is possible to have one condition be optimized and the other not. This is known as a partially-optimized query.

How DBISAM Builds the Query Results

Once an index is selected for optimizing a given condition of the WHERE clause, a range is set on the index in order to limit the index keys to those that match the current condition being optimized. The index keys that satisfy the condition are then scanned, and during the scan a bitmap is built in physical record number order. A bit is turned on if the physical record satisfies the condition, and a bit is turned off if it doesn't. This method of using bitmaps works well because it can represent sets of data with minimal memory consumption. Also, DBISAM is able to quickly determine how many records are in the set (how many bits are turned on), and it can easily AND, OR, and NOT bitmaps together to fulfill boolean logic between multiple conditions. Finally, because the bitmap is in physical record order, accessing the records using a bitmap is very direct since DBISAM uses fixed-length records with directly-addressable offsets in the physical table format.

When optimizing SQL SELECT queries that contain both join conditions and WHERE conditions, DBISAM always processes the non-join conditions first if the conditions do not affect the target table, which is the table on the right side of a LEFT OUTER JOIN or the table on the left side of a RIGHT OUTER JOIN. This can speed up join operations tremendously since the join conditions will only take into account the records existing in the source table(s) based upon the WHERE conditions. For example, consider the following query:

```
SELECT
OrderHdr.Cust_ID,
OrderHdr.Order_Num,
```

```
OrderDet.Model_Num,
OrderDet.Cust_Item
FROM OrderHdr, OrderDet
WHERE OrderHdr.Order_Num=OrderDet.Order_Num AND
      OrderHdr.Cust_ID='C901'
ORDER BY 1,2,3
```

In this example, the WHERE condition:

```
OrderHdr.Cust_ID='C901'
```

will be evaluated first before the join condition:

```
OrderHdr.Order_Num=OrderDet.Order_Num
```

so that the joins only need to process a small number of records in the OrderHdr table.

When optimizing SQL SELECT queries that contain INNER JOINs that also contain selection conditions (conditions in an INNER JOIN clause that do not specify an actual join), the selection conditions are always processed at the same time as the join, even if they affect the target table, which is the table on the right side of the join. This can speed up join operations tremendously since the join conditions will only take into account the records existing in the source table(s) based upon the selection conditions. For example, consider the following query:

```
SELECT
OrderHdr.Cust_ID,
OrderHdr.Order_Num,
OrderDet.Model_Num,
OrderDet.Cust_Item
FROM OrderHdr INNER JOIN OrderDet ON
OrderHdr.Order_Num=OrderDet.Order_Num AND OrderHdr.Cust_ID='C901'
ORDER BY 1,2,3
```

In this example, the selection condition:

```
OrderHdr.Cust_ID='C901'
```

will be evaluated first before the join condition:

```
OrderHdr.Order_Num=OrderDet.Order_Num
```

so that the joins only need to process a small number of records in the OrderHdr table.

Note
If an SQL SELECT query can return a live result set, then the WHERE clause conditions are applied to the source table via an optimized filter and the table is opened. If an SQL SELECT query contains joins or other items that cause DBISAM to only return a canned result set, then all of the records from the source tables that satisfy the WHERE clause conditions and join conditions are copied to a temporary table on disk and that table is opened as the query result set. This process can be time-consuming when a large number of records are returned by the query, so it is recommended that you try to make your queries as selective as possible.

How Joins are Processed

Join conditions in SQL SELECT, UPDATE, or DELETE statements are processed in DBISAM using a technique known as nested-loop joins. This means that DBISAM recursively processes the source tables in a master-detail, master-detail, etc. arrangement with a driving table and a destination table (which then becomes the driving table for any subsequent join conditions). When using this technique, it is very important that the table with the smallest record count (after any non-join conditions from the WHERE clause have been applied) is specified as the first driving table in the processing of the joins. DBISAM's SQL optimizer will automatically optimize the join ordering so that the table with the smallest record count is placed as the first driving table as long as the joins are INNER JOINS or SQL-89 joins in the WHERE clause. LEFT OUTER JOINs and RIGHT OUTER JOINs cannot be re-ordered in such a fashion and must be left alone.

The following is an example that illustrates the nested-loop joins in DBISAM:

```
SELECT c.Company,
o.OrderNo,
e.LastName,
p.Description,
v.VendorName
FROM Customer c, Orders o, Items i, Vendors v, Parts p, Employee e
WHERE c.CustNo=o.CustNo AND
o.OrderNo=i.OrderNo AND
i.PartNo=p.PartNo AND
p.VendorNo=v.VendorNo AND
o.EmpNo=e.EmpNo
ORDER BY e.LastName
```

In this example, DBISAM would process the joins in this order:

1) Customer table joined to Orders table on the CustNo column

2) Orders table joined to Items table on the OrderNo column and Orders table joined to Employee table on EmpNo column (this is also known as a multi-way, or star, join)

3) Items table joined to Parts table on the PartNo column

4) Parts table joined to Vendors table on the VendorNo column

In this case the Customer table is the smallest table in terms of record count, so making it the driving table in this case is a good choice. Also, you'll notice that in the case of the multi-way, or star, join between the Orders table and both the Items and Employee table, DBISAM will move the join order of the Employee table up in order to keep the join ordering as close to the order of the source tables in the

FROM clause as possible.

Note
You can use the NOJOINOPTIMIZE keyword at the end of the SQL SELECT, UPDATE, or DELETE statement in order to tell DBISAM not to reorder the joins. Also, SQL UPDATE and DELETE statements cannot have their driver table reordered due to the fact that the driver table is the table being updated by these statements.

Query Plans

You can use the TDBISAMQuery GeneratePlan property to indicate that you want DBISAM to generate a query plan for the current SQL statement or script when it is executed. The resulting query plan will be stored in the TDBISAMQuery Plan property. Examining this query plan can tell you exactly what the SQL optimizer is doing when executing a given SQL statement or script. For example, the query mentioned above would generate the following query plan:

```
================================================================================
      ===
SQL
      statement
================================================================================
      ==

SELECT c.Company,
o.OrderNo,
e.LastName,
p.Description,
v.VendorName
FROM Customer c, Orders o, Items i, Vendors v, Parts p, Employee e
WHERE c.CustNo=o.CustNo AND
o.OrderNo=i.OrderNo AND
i.PartNo=p.PartNo AND
p.VendorNo=v.VendorNo AND
o.EmpNo=e.EmpNo
ORDER BY e.LastName

Result Set Generation
---------------------

Result set will be canned
Result set will consist of one or more rows
Result set will be ordered by the following column(s) using a case-sensitive
temporary index:

LastName ASC

Join Ordering
-------------

The driver table is the Customer table (c)

The Customer table (c) is joined to the Orders table (o) with the INNER JOIN
expression:
```

```
c.CustNo = o.CustNo

The Orders table (o) is joined to the Items table (i) with the INNER JOIN
expression:

o.OrderNo = i.OrderNo

The Orders table (o) is joined to the Employee table (e) with the INNER JOIN
expression:

o.EmpNo = e.EmpNo

The Items table (i) is joined to the Parts table (p) with the INNER JOIN
expression:

i.PartNo = p.PartNo

The Parts table (p) is joined to the Vendors table (v) with the INNER JOIN
expression:

p.VendorNo = v.VendorNo

Optimized Join Ordering
-----------------------

The driver table is the Vendors table (v)

The Vendors table (v) is joined to the Parts table (p) with the INNER JOIN
expression:

v.VendorNo = p.VendorNo

The Parts table (p) is joined to the Items table (i) with the INNER JOIN
expression:

p.PartNo = i.PartNo

The Items table (i) is joined to the Orders table (o) with the INNER JOIN
expression:

i.OrderNo = o.OrderNo

The Orders table (o) is joined to the Customer table (c) with the INNER JOIN
expression:

o.CustNo = c.CustNo

The Orders table (o) is joined to the Employee table (e) with the INNER JOIN
expression:

o.EmpNo = e.EmpNo

Join Execution
--------------

Costs ARE NOT being taken into account when executing this join
Use the JOINOPTIMIZECOSTS clause at the end of the SQL statement to force the

optimizer to consider costs when optimizing this join
```

```
The expression:

v.VendorNo = p.VendorNo

is OPTIMIZED

The expression:

p.PartNo = i.PartNo

is OPTIMIZED

The expression:

i.OrderNo = o.OrderNo

is OPTIMIZED

The expression:

o.CustNo = c.CustNo

is OPTIMIZED

The expression:

o.EmpNo = e.EmpNo

is
      OPTIMIZED

================================================================================
      ==
```

You'll notice that the joins have been re-ordered to be in the most optimal order. You'll also notice that the query plan mentions that the JOINOPTIMIZECOSTS clause is not being used. Use a JOINOPTIMIZECOSTS clause to force the query optimizer to use I/O cost projections to determine the most efficient way to process the conditions in a join expression. If you have a join expression with multiple conditions in it, then using this clause may help improve the performance of the join expression, especially if it is already executing very slowly.

Further Optimizations Provided by DBISAM

In addition to just using indexes to speed up the querying process, DBISAM also provides a few other optimizations that can greatly increase a given query's performance. When building a bitmap for a given optimized condition, DBISAM can take advantage of statistics that are kept in DBISAM indexes. These statistics accurately reflect the current make-up of the various values present in the index, and DBISAM uses this information to optimize the actual scan of the index.

DBISAM looks at the optimization of the query conditions, and when multiple conditions are joined by an AND operator, DBISAM ensures that the most optimized query condition is executed first. For example, consider a table of 25,000 records with the following structure:

```
Customer table
```

```
Field       Data Type      Index
-------------------------------------------------
ID          Integer        Primary Index
Name        String[30]
State       String[2]      Secondary, case-sensitive,
                           non-unique, ascending, index
TotalOrders BCD[2]
```

And consider the following SQL SELECT query:

```
SELECT *
FROM customer
WHERE (TotalOrders > 10000) and (State='CA')
```

As you can see, the TotalOrders condition cannot be optimized since no indexes exist that would allow for optimization, whereas the State condition can be optimized. If only 200 records in the table have a State field that contains 'CA', then processing the query in the order indicated by the expression would be very inefficient, since the following steps would take place:

1) All 25,000 physical records would be read and evaluated to build a bitmap for the (TotalOrders > 10000) condition.

2) The resultant bitmap from the previous step would be ANDed together with a bitmap built using the optimized index scan for the State condition.

DBISAM uses a much better approach because it knows that:

1) The TotalOrders condition is not optimized

2) The State condition is optimized

3) Both conditions are joined using the AND operator

it is able to reverse the query conditions in the WHERE clause and execute the index scan for the 200 records that satisfy the State condition first, and then proceed to only read the 200 records from disk in order to evaluate the TotalOrders condition. DBISAM has just saved a tremendous amount of I/O by simply reversing the query conditions.

Note
This optimization only works with query conditions that are joined by the AND operator. If the above two conditions were joined using the OR operator, then DBISAM would simply read all 25,000 records and evaluate the entire WHERE expression for each record.

In the case of a completely un-optimized query, DBISAM's read-ahead buffering can help tremendously in reducing network traffic and providing the most efficient reads with the least amount of I/O calls to the operating system. DBISAM will read up to 32 kilobytes of contiguous records on disk in the course of processing an un-optimized query.

DBISAM can also optimize for the UPPER() and LOWER() SQL functions by using any case-insensitive indexes in the source tables to optimize the query condition. Take the following table for example:

```
Customer table

Field       Data Type      Index
-------------------------------------------------
ID          Integer        Primary Index
Name        String[30]
State       String[2]      Secondary, case-insensitive,
                           non-unique, ascending, index
```

And consider the following SQL SELECT query:

```
SELECT *
FROM customer
WHERE (UPPER(State)='CA')
```

In this query, DBISAM will be able to select and use the case-insensitive index on the State field, and this is caused by the presence of the UPPER() function around the field name. This can also be used to optimize joins. For example, here are two tables that use case-insensitive indexes for optimizing joins:

```
Customer table

Field       Data Type      Index
-------------------------------------------------
ID          String[10]     Primary, case-insensitive
                           index
Name        String[30]
State       String[2]
```

```
Orders table

Field       Data Type      Index
-------------------------------------------------
OrderNum    String[20]     Primary, case-insensitive
                           index
CustID      String[10]     Secondary, case-insensitive
                           index
TotalAmount BCD[2]
```

And consider the following SQL SELECT query:

```
SELECT *
FROM Customer, Orders
WHERE (UPPER(Customer.ID)=UPPER(Orders.CustID))
```

In this query, the join condition will be optimized due to the presence of the UPPER() function around the Orders.CustID field. The UPPER() function around the Customer.ID field is simply to ensure that the join is

made on upper-case customer ID values only.

Optimization Levels

DBISAM determines the level of optimization for a WHERE or JOIN clause using the following rules:

```
Optimized Condition = Fully-Optimized WHERE or JOIN clause

Un-Optimized Condition = Un-Optimized WHERE or JOIN clause

Optimized Condition AND Optimized Condition = Fully-
Optimized WHERE or JOIN clause

Optimized Condition AND Un-Optimized Condition = Partially-
Optimized WHERE or JOIN clause

Un-Optimized Condition AND Optimized Condition = Partially-
Optimized WHERE or JOIN clause

Un-Optimized Condition AND Un-Optimized Condition = Un-
Optimized WHERE or JOIN clause

Optimized Condition OR Optimized Condition = Fully-
Optimized WHERE or JOIN clause

Optimized Condition OR Un-Optimized Condition = Un-
Optimized WHERE or JOIN clause

Un-Optimized Condition OR Optimized Condition = Un-
Optimized WHERE or JOIN clause

Un-Optimized Condition OR Un-Optimized Condition = Un-
Optimized WHERE or JOIN clause
```

Note
The unary NOT operator causes any expression to become partially optimized. This is due to the fact that DBISAM must scan for, and remove, deleted records from the current records bitmap once it has taken the bitmap and performed the NOT operation on the bits.

DBISAM Limitations

DBISAM does not optimize multiple query conditions joined by an AND operator) by mapping them to a compound index that may be available. To illustrate this point, consider a table with the following structure:

```
Employee

Field       Data Type      Index
------------------------------------------------------------------------
LastName    String[30]     Primary Index  (both fields are part of the
FirstName   String[20]     Primary Index   primary index)
```

And consider the following query:

```
SELECT *
FROM Employee
WHERE (LastName='Smith') and (FirstName='John')
```

Logically you would assume that DBISAM can use the one primary index in order to optimize the entire WHERE clause. Unfortunately this is not the case, and instead DBISAM will only use the primary index for optimizing the LastName condition and resort to reading records in order to evaluate the FirstName condition.

4.5 Operators

Introduction

DBISAM allows comparison operators, extended comparison operators, arithmetic operators, string operators, date, time, and timestamp operators, and logical operators in SQL statements. These operators are detailed below.

Comparison Operators

Use comparison operators to perform comparisons on data in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following comparison operators:

Operator	Description
<	Determines if a value is less than another value.
>	Determines if a value is greater than another value.
=	Determines if a value is equal to another value.
<>	Determines if a value is not equal to another value.
>=	Determines if a value is greater than or equal to another value.
<=	Determines if a value is less than or equal to another value.

Use comparison operators to compare two like values. Values compared can be: column values, literals, or calculations. The result of the comparison is a boolean value that is used in contexts like a WHERE clause to determine on a row-by-row basis whether a row meets the filtering criteria. The following example uses the >= comparison operator to show only the orders where the ItemsTotal column is greater than or equal to 1000:

```
SELECT *
FROM Orders
WHERE (ItemsTotal >= 1000)
```

Comparisons must be between two values of the same or a compatible data type. The result of a comparison operation can be modified by a logical operator, such as NOT. The following example uses the >= comparison operator and the logical NOT operator to show only the orders where the ItemsTotal column is not greater than or equal to 1000:

```
SELECT *
FROM Orders
WHERE NOT (ItemsTotal >= 1000)
```

Note
Comparison operators can only be used in a WHERE or HAVING clause, or in the ON clause of a join - they cannot be used in the SELECT clause. The only exception to this would be within the first argument to the IF() function, which allows comparison expressions for performing IF...ELSE boolean logic.

Extended Comparison Operators

Use extended comparison operators to perform comparisons on data in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM supports the following extended comparison operators:

Operator	Description
[NOT] BETWEEN	Compares a value to a range formed by two values.
[NOT] IN	Determines whether a value exists in a list of values.
[NOT] LIKE	Compares, in part or in whole, one value with another.
IS [NOT] NULL	Compares a value with an empty, or NULL, value.

BETWEEN Extended Comparison Operator

The BETWEEN extended comparison operator determines whether a value falls inside a range. The syntax is as follows:

```
value1 [NOT] BETWEEN value2 AND value3
```

Use the BETWEEN extended comparison operator to compare a value to a value range. If the value is greater than or equal to the low end of the range and less than or equal to the high end of the range, BETWEEN returns a TRUE value. If the value is less than the low end value or greater than the high end value, BETWEEN returns a FALSE value. For example, the expression below returns a FALSE value because 10 is not between 1 and 5:

```
10 BETWEEN 1 AND 5
```

Use NOT to return the converse of a BETWEEN comparison. For example, the expression below returns a TRUE value:

```
10 NOT BETWEEN 1 AND 5
```

BETWEEN can be used with all non-BLOB data types, but all values compared must be of the same or a compatible data type. The left-side and right-side values used in a BETWEEN comparison may be columns, literals, or calculated values. The following example returns all orders where the SaleDate column is between January 1, 1998 and December 31, 1998:

```
SELECT SaleDate
FROM Orders
WHERE (SaleDate BETWEEN '1998-01-01' AND '1998-12-31')
```

BETWEEN is useful when filtering to retrieve rows with contiguous values that fall within the specified range. For filtering to retrieve rows with noncontiguous values, use the IN extended comparison operator.

IN Extended Comparison Operator

The IN extended comparison operator indicates whether a value exists in a set of values. The syntax is as follows:

```
value [NOT] IN (value_set)
```

Use the IN extended comparison operator to filter a table based on the existence of a column value in a specified set of comparison values. The set of comparison values can be a comma-separated list of column names, literals, or calculated values. The following example returns all customers where the State column is either 'CA' or 'HI':

```
SELECT c.Company, c.State
FROM Customer c
WHERE (c.State IN ('CA', 'HI'))
```

The value to compare with the values set can be any or a combination of a column value, a literal value, or a calculated value. Use NOT to return the converse of an IN comparison. IN can be used with all non-BLOB data types, but all values compared must be of the same or a compatible data type.

IN is useful when filtering to retrieve rows with noncontiguous values. For filtering to retrieve rows with contiguous values that fall within a specified range, use the BETWEEN extended comparison operator.

LIKE Extended Comparison Operator

The LIKE extended comparison operator indicates the similarity of one value as compared to another. The syntax is as follows:

```
value [NOT] LIKE [substitution_char] comparison_value
[substitution_char] ESCAPE escape_char
```

Use the LIKE extended comparison operator to filter a table based on the similarity of a column value to a comparison value. Use of substitution characters allows the comparison to be based on the whole column value or just a portion. The following example returns all customers where the Company column is equal to 'Adventure Undersea':

```
SELECT *
FROM Customer
WHERE (Company LIKE 'Adventure Undersea')
```

The wildcard substitution character (%) may be used in the comparison to represent an unknown number of characters. LIKE returns a TRUE when the portion of the column value matches that portion of the comparison value not corresponding to the position of the wildcard character. The wildcard character can appear at the beginning, middle, or end of the comparison value (or multiple combinations of these positions). The following example retrieves rows where the column value begins with 'A' and is followed by any number of any characters. Matching values could include 'Action Club' and 'Adventure Undersea', but not 'Blue Sports':

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

The single-character substitution character (_) may be used in the comparison to represent a single character. LIKE returns a TRUE when the portion of the column value matches that portion of the comparison value not corresponding to the position of the single-character substitution character. The single-character substitution character can appear at the beginning, middle, or end of the comparison value (or multiple combinations of these positions). Use one single-character substitution character for each character to be wild in the filter pattern. The following example retrieves rows where the column value begins with 'b' ends with 'n', with one character of any value between. Matching values could include 'bin' and 'ban', but not 'barn':

```
SELECT Words
FROM Dictionary
WHERE (Words LIKE 'b_n')
```

The ESCAPE keyword can be used after the comparison to represent an escape character in the comparison value. When an escape character is found in the comparison value, DBISAM will treat the next character after the escape character as a literal and not a wildcard character. This allows for the use of the special wildcard characters as literal search characters in the comparison value. For example, the following example retrieves rows where the column value contains the string constant '10%':

```
SELECT ID, Description
FROM Items
WHERE (Description LIKE '%10\%%') ESCAPE '\'
```

Use NOT to return the converse of a LIKE comparison. LIKE can be used only with string or compatible data types such as memo columns. The comparison performed by the LIKE extended comparison operator is always case-sensitive.

IS NULL Extended Comparison Operator

The IS NULL extended comparison operator indicates whether a column contains a NULL value. The syntax is as follows:

```
column_reference IS [NOT] NULL
```

Use the IS NULL extended comparison operator to filter a table based on the specified column containing a NULL (empty) value. The following example returns all customers where the InvoiceDate column is null:

```
SELECT *
FROM Customer
WHERE (InvoiceDate IS NULL)
```

Use NOT to return the converse of a IS NULL comparison.

Note
For a numeric column, a zero value is not the same as a NULL value.

Value Operators

Use value operators to return specific values based upon other expressions in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM supports the following value operators:

Operator	Description
CASE	Evaluates a series of boolean expressions and returns the matching result value.

CASE Value Operator

The CASE value operator can be used in with two different syntaxes, one being the normal syntax while the other being a shorthand syntax. The normal syntax is used to evaluate a series of boolean expressions and return the matching result value for the first boolean expression that returns True, and is as follows:

```
CASE
WHEN boolean expression THEN value
[WHEN boolean expression THEN value]
[ELSE] value
END
```

The following is an example of the normal CASE syntax. It translate a credit card type into a more verbose description:

```
SELECT CardType,
CASE
WHEN Upper(CardType)='A' THEN 'American Express'
WHEN Upper(CardType)='M' THEN 'Mastercard'
WHEN Upper(CardType)='V' THEN 'Visa'
WHEN Upper(CardType)='D' THEN 'Diners Club'
END AS CardDesc,
SUM(SalesAmount) AS TotalSales
FROM Transactions
GROUP BY CardType
```

```
ORDER BY TotalSales DESC
```

The shorthand syntax is as follows:

```
CASE expression
WHEN expression THEN value
[WHEN expression THEN value]
[ELSE] value
END
```

The primary difference between the shorthand syntax and the normal syntax is the inclusion of the expression directly after the CASE operator itself. It is used as the comparison value for every WHEN expression. All WHEN expressions must be type-compatible with this expression and can be any type, unlike the normal syntax which requires boolean expressions. The rest of the shorthand syntax is the same as the normal syntax.

The following is the above credit card type example using the shorthand syntax:

```
SELECT CardType,
CASE Upper(CardType)
WHEN 'A' THEN 'American Express'
WHEN 'M' THEN 'Mastercard'
WHEN 'V' THEN 'Visa'
WHEN 'D' THEN 'Diners Club'
END AS CardDesc,
SUM(SalesAmount) AS TotalSales
FROM Transactions
GROUP BY CardType
ORDER BY TotalSales DESC
```

Arithmetic Operators

Use arithmetic operators to perform arithmetic calculations on data in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following arithmetic operators:

Operator	Description
+	Add two numeric values together numeric value.
-	Subtract one numeric value from another numeric value.
*	Multiply one numeric value by another numeric value.
/	Divide one numeric value by another numeric value.
MOD	Returns the modulus of the two integer arguments as an integer

Calculations can be performed wherever non-aggregated data values are allowed, such as in a SELECT or WHERE clause. In following example, a column value is multiplied by a numeric literal:

```
```

```
SELECT (itemstotal * 0.0825) AS Tax
FROM orders
```

Arithmetic calculations are performed in the normal order of precedence: multiplication, division, modulus, addition, and then subtraction. To cause a calculation to be performed out of the normal order of precedence, use parentheses around the operation to be performed first. In the next example, the addition is performed before the multiplication:

```
SELECT (n.numbers * (n.multiple + 1)) AS Result
FROM numbertable n
```

Arithmetic operators operate only on numeric values.

String Operators

Use string operators to perform string concatenation on character data in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following string operators:

Operator	Description
+	Concatenate two string values together.
||	Concatenate two string values together.

String operations can be performed wherever non-aggregated data values are allowed, such as in a SELECT or WHERE clause. In following example, a column value concatenated with a second column value to provide a new calculated column in the query result set:

```
SELECT (LastName + ', ' + FirstName) AS FullName
FROM Employee
```

String operators operate only on string values or memo columns.

Date, Time, and Timestamp Operators

Use date, time, and timestamp operators to perform date, time, and timestamp calculations in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following date, time, and timestamp operators:

Operator	Description
+	Adding days or milliseconds to date, time, or timestamp values.
-	Subtracting days or milliseconds from date, time, or timestamp values, or subtracting two date, time, or timestamp values to get the difference in days or millseconds.

The rules for adding or subtracting dates, times, and timestamps in conjunction with integers are as follows:

Adding an integer to a date is equivalent to adding days to the date

Adding an integer to a time is equivalent to adding milliseconds to the time (be careful of wraparound since a time value is equal to the number of milliseconds elapsed since the beginning of the current day)

Adding an integer to a timestamp is equivalent to adding milliseconds to the time portion of the timestamp (any milliseconds beyond the number of milliseconds in a day will result in an increment of the day value in the timestamp by 1)

Subtracting an integer from a date is equivalent to subtracting days from the date

Subtracting an integer from a time is equivalent to subtracting milliseconds from the time (be careful of going below 0, which will be ignored)

Subtracting an integer from a timestamp is equivalent to subtracting milliseconds from the time portion of the timestamp (any milliseconds less than 0 for the time portion will result in a decrement of the day value in the timestamp by 1)

Subtracting a date value from another date value will result in the number of days between the two dates (be sure to use the ABS() function to ensure a positive value if the second value is larger than the first)

Subtracting a time value from another time value will result in the number of milliseconds between the two times (be sure to use the ABS() function to ensure a positive value if the second value is larger than the first)

Subtracting a date value from a timestamp value will result in the number of milliseconds between the timestamp and the date (be sure to use the ABS() function to ensure a positive value if the second value is larger than the first)

Subtracting a timestamp value from a timestamp value will result in the number of milliseconds between the timestamp and the other timestamp (be sure to use the ABS() function to ensure a positive value if the second value is larger than the first)

The following example shows how you would add 30 days to a date to get an invoice due date for an invoice in a SELECT SQL statement:

```
SELECT InvoiceDate, (InvoiceDate + 30) AS DueDate, BalanceDue
FROM Invoices
WHERE InvoiceDate BETWEEN '1999-01-01' AND '1999-01-31'
```

Date, time, and timestamp operators operate only on date, time, or timestamp values in conjuction with integer values.

Logical Operators

Use logical operators to perform Boolean logic between different predicates (conditions) in an SQL WHERE clause. DBISAM's SQL supports the following logical operators:

Operator	Description

NOT	NOT a boolean value.
AND	AND two boolean values together.
OR	OR two boolean values together.

This allows the source table(s) to be filtered based on multiple conditions. Logical operators compare the boolean result of two predicate comparisons, each producing a boolean result. If OR is used, either of the two predicate comparisons can result on a TRUE value for the whole expression to evaluate to TRUE. If AND is used, both predicate comparisons must evaluate to TRUE for the whole expression to be TRUE; if either is FALSE, the whole is FALSE. In the following example, if only one of the two predicate comparisons is TRUE, the row will be included in the query result set:

```
SELECT *
FROM Reservations
WHERE ((ReservationDate < '1998-01-31') OR (Paid = TRUE))
```

Logical operator comparisons are performed in the order of AND and then OR. To perform a comparison out of the normal order of precedence, use parentheses around the comparison to be performed first. The SELECT statement below retrieves all rows where the Shape column is 'Round' and the Color 'Blue':

```
SELECT Shape, Color
FROM Objects
WHERE (Color = 'Red' OR Shape = 'Round') AND Color = 'Blue'
```

Without the parentheses, the default order of precedence is used and the logic changes. The next example, a variation on the above statement, would return rows where the Shape is 'Round' and the Color is 'Blue', but would also return rows where the Color is 'Red', regardless of the Shape:

```
SELECT Shape, Color
FROM Objects
WHERE Color = 'Red' OR Shape = 'Round' AND Color = 'Blue'
```

Use the NOT operator to negate the boolean result of a comparison. In the following example, only those rows where the Paid column contains a FALSE value are retrieved:

```
SELECT *
FROM reservations
WHERE (NOT (Paid = TRUE))
```

4.6 Functions

Introduction

DBISAM's SQL provides string functions, numeric functions, boolean functions, aggregate functions (used in conjunction with an SQL SELECT GROUP BY clause), autoinc functions, full text indexing functions, and data conversion functions.

String Functions

Use string functions to manipulate string values in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following string functions:

Function	Description
LOWER or LCASE	Forces a string to lowercase.
UPPER or UCASE	Forces a string to uppercase.
LENGTH	Returns the length of a string value.
SUBSTRING	Extracts a portion of a string value.
LEFT	Extracts a certain number of characters from the left side of a string value.
RIGHT	Extracts a certain number of characters from the right side of a string value.
TRIM	Removes repetitions of a specified character from the left, right, or both sides of a string.
LTRIM	Removes any leading space characters from a string.
RTRIM	Removes any trailing space characters from a string.
POS or POSITION	Finds the position of one string value within another string value.
OCCURS	Finds the number of times one string value is present within another string value.
REPLACE	Replaces all occurrences of one string value with a new string value within another string value.
REPEAT	Repeats a string value a specified number of times.
CONCAT	Concatenates two string values together.

LOWER or LCASE Function

The LOWER or LCASE function converts all characters in a string value to lowercase. The syntax is as follows:

```
LOWER(column_reference or string constant)
LCASE(column_reference or string constant)
```

In the following example, the values in the NAME column appear all in lowercase:

```
SELECT LOWER(Name)
FROM Country
```

The LOWER or LCASE function can be used in WHERE clause string comparisons to cause a case-insensitive comparison. Apply LOWER or LCASE to the values on both sides of the comparison operator (if one of the comparison values is a literal, simply enter it all in lower case).

```
SELECT *
FROM Names
WHERE LOWER(Lastname) = 'smith'
```

LOWER or LCASE can only be used with string or memo columns or constants.

UPPER or UCASE Function

The UPPER or UCASE function converts all characters in a string value to uppercase. The syntax is as follows:

```
UPPER(column_reference or string constant)
UCASE(column_reference or string constant)
```

Use UPPER or UCASE to convert all of the characters in a table column or character literal to uppercase. In the following example, the values in the NAME column are treated as all in uppercase. Because the same conversion is applied to both the filter column and comparison value in the WHERE clause, the filtering is effectively case-insensitive:

```
SELECT Name, Capital, Continent
FROM Country
WHERE UPPER(Name) LIKE UPPER('PE%')
```

UPPER can only be used with string or memo columns or constants.

LENGTH Function

The LENGTH function returns the length of a string value as an integer value. The syntax is as follows:

```
LENGTH(column_reference or string constant)
```

In the following example, the length of the values in the Notes column are returned as part of the SELECT statement:

```
SELECT Notes, LENGTH(Notes) AS "Num Chars"
FROM Biolife
```

LENGTH can only be used with string or memo columns or constants.

SUBSTRING Function

The SUBSTRING function extracts a substring from a string. The syntax is as follows:

```
SUBSTRING(column_reference or string constant
          FROM start_index [FOR length])
SUBSTRING(column_reference or string constant,
          start_index[,length])
```

The second FROM parameter is the character position at which the extracted substring starts within the original string. The index for the FROM parameter is based on the first character in the source value being 1.

The FOR parameter is optional, and specifies the length of the extracted substring. If the FOR parameter is omitted, the substring goes from the position specified by the FROM parameter to the end of the string.

In the following example, the SUBSTRING function is applied to the literal string 'ABCDE' and returns the value 'BCD':

```
SELECT SUBSTRING('ABCDE' FROM 2 FOR 3) AS Sub
FROM Country
```

In the following example, only the second and subsequent characters of the NAME column are retrieved:

```
SELECT SUBSTRING(Name FROM 2)
FROM Country
```

SUBSTRING can only be used with string or memo columns or constants.

LEFT Function

The LEFT function extracts a certain number of characters from the left side of a string. The syntax is as follows:

```
LEFT(column_reference or string constant FOR length)
LEFT(column_reference or string constant,length)
```

The FOR parameter specifies the length of the extracted substring.

In the following example, the LEFT function is applied to the literal string 'ABCDE' and returns the value 'ABC':

```
SELECT LEFT('ABCDE' FOR 3) AS Sub
FROM Country
```

LEFT can only be used with string or memo columns or constants.

RIGHT Function

The RIGHT function extracts a certain number of characters from the right side of a string. The syntax is as follows:

```
RIGHT(column_reference or string constant FOR length)
RIGHT(column_reference or string constant,length)
```

The FOR parameter specifies the length of the extracted substring.

In the following example, the RIGHT function is applied to the literal string 'ABCDE' and returns the value 'DE':

```
SELECT RIGHT('ABCDE' FOR 2) AS Sub
FROM Country
```

RIGHT can only be used with string or memo columns or constants.

TRIM Function

The TRIM function removes the trailing or leading character, or both, from a string. The syntax is as follows:

```
TRIM([LEADING|TRAILING|BOTH] trimmed_char
     FROM column_reference or string constant)
TRIM([LEADING|TRAILING|BOTH] trimmed_char,
     column_reference or string constant)
```

The first parameter indicates the position of the character to be deleted, and has one of the following values:

Keyword	Description
LEADING	Deletes the character at the left end of the string.
TRAILING	Deletes the character at the right end of the string.
BOTH	Deletes the character at both ends of the string.

The trimmed character parameter specifies the character to be deleted. Case-sensitivity is applied for this parameter. To make TRIM case-insensitive, use the UPPER or UCASE function on the column reference or string constant.

The FROM parameter specifies the column or constant from which to delete the character. The column reference for the FROM parameter can be a string column or a string constant.

The following are examples of using the TRIM function:

```
TRIM(LEADING '_' FROM '_ABC_') will return 'ABC_'
TRIM(TRAILING '_' FROM '_ABC_') will return '_ABC'
TRIM(BOTH '_' FROM '_ABC_') will return 'ABC'
TRIM(BOTH 'A' FROM 'ABC') will return 'BC'
```

TRIM can only be used with string or memo columns or constants.

LTRIM Function

The LTRIM function removes any leading spaces from a string. The syntax is as follows:

```
LTRIM(column_reference or string constant)
```

The first and only parameter specifies the column or constant from which to delete the leading spaces, if any are present. The following is an example of using the LTRIM function:

```
LTRIM('   ABC') will return 'ABC'
```

LTRIM can only be used with string or memo columns or constants.

RTRIM Function

The RTRIM function removes any trailing spaces from a string. The syntax is as follows:

```
RTRIM(column_reference or string constant)
```

The first and only parameter specifies the column or constant from which to delete the trailing spaces, if any are present. The following is an example of using the RTRIM function:

```
RTRIM('ABC   ') will return 'ABC'
```

RTRIM can only be used with string or memo columns or constants.

POS or POSITION Function

The POS or POSITION function returns the position of one string within another string. The syntax is as follows:

```
POS(string constant IN column_reference or string constant)
POSITION(string constant IN column_reference or string constant)
POS(string constant,column_reference or string constant)
POSITION(string constant,column_reference or string constant)
```

If the search string is not present, then 0 will be returned.

In the following example, the POS function is used to select all rows where the literal string 'ABC' exists in the Name column:

```
SELECT *
FROM Country
WHERE POS('ABC' IN Name) > 0
```

POS or POSITION can only be used with string or memo columns or constants.

OCCURS Function

The OCCURS function returns the number of occurrences of one string within another string. The syntax is as follows:

```
OCCURS(string constant
       IN column_reference or string constant)
OCCURS(string constant,
       column_reference or string constant)
```

If the search string is not present, then 0 will be returned.

In the following example, the OCCURS function is used to select all rows where the literal string 'ABC' occurs at least once in the Name column:

```
SELECT *
FROM Country
WHERE OCCURS('ABC' IN Name) > 0
```

OCCURS can only be used with string or memo columns or constants.

REPLACE Function

The REPLACE function replaces all occurrences of a given string with a new string within another string. The syntax is as follows:

```
REPLACE(string constant WITH new string constant
        IN column_reference or string constant)
REPLACE(string constant,new string constant,
        column_reference or string constant)
```

If the search string is not present, then the result will be the original table column or string constant.

In the following example, the REPLACE function is used to replace all occurrences of 'Mexico' with 'South America':

```
UPDATE biolife
SET notes=REPLACE('Mexico' WITH 'South America' IN notes)
```

REPLACE can only be used with string or memo columns or constants.

REPEAT Function

The REPEAT function repeats a given string a specified number of times and returns the concatenated result. The syntax is as follows:

```
REPEAT(column_reference or string constant
       FOR number_of_occurrences)
REPEAT(column_reference or string constant,
       number_of_occurrences)
```

In the following example, the REPEAT function is used to replicate the dash (-) character 60 times to use as a separator in a multi-line string:

```
UPDATE biolife
SET notes='Notes'+#13+#10+
REPEAT('-' FOR 60)+#13+#10+#13+#10+
'These are the notes'
```

REPEAT can only be used with string or memo columns or constants.

CONCAT Function

The CONCAT function concatenates two strings together and returns the concatenated result. The syntax is as follows:

```
CONCAT(column_reference or string constant
       WITH column_reference or string constant)
CONCAT(column_reference or string constant,
       column_reference or string constant)
```

In the following example, the CONCAT function is used to concatenate two strings together:

```
UPDATE biolife
SET notes=CONCAT(Notes WITH #13+#10+#13+#10+'End of Notes')
```

CONCAT can only be used with string or memo columns or constants.

Numeric Functions

Use numeric functions to manipulate numeric values in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following numeric functions:

Function	Description
ABS	Converts a number to its absolute value (non-negative).
ACOS	Returns the arccosine of a number as an angle expressed in radians.
ASIN	Returns the arcsine of a number as an angle expressed in radians.
ATAN	Returns the arctangent of a number as an angle expressed in radians.
ATAN2	Returns the arctangent of x and y coordinates as an angle expressed in radians.
CEIL or CEILING	Returns the lowest integer greater than or equal to a number.
COS	Returns the cosine of an angle.
COT	Returns the cotangent of an angle.
DEGREES	Converts a number representing radians into degrees.
EXP	Returns the exponential value of a number.
FLOOR	Returns the highest integer less than or equal to a number.
LOG	Returns the natural logarithm of a number.
LOG10	Returns the base 10 logarithm of a number.
MOD	Returns the modulus of two integers as an integer.
PI	Returns the ratio of a circle's circumference to its diameter - approximated as 3.1415926535897932385.
POWER	Returns the value of a base number raised to the specified power.
RADIANS	Converts a number representing degrees into radians.
RAND	Returns a random number.
ROUND	Rounds a number to a specified number of decimal places.
SIGN	Returns -1 if a number is less than 0, 0 if a number is 0, or 1 if a number is greater than 0.
SIN	Returns the sine of an angle.

SQRT	Returns the square root of a number.
TAN	Returns the tangent of an angle.
TRUNC or TRUNCATE	Truncates a numeric argument to the specified number of decimal places

ABS Function

The ABS function converts a numeric value to its absolute, or non-negative value:

```
ABS(column_reference or numeric constant)
```

ABS can only be used with numeric columns or constants.

ACOS Function

The ACOS function returns the arccosine of a number as an angle expressed in radians:

```
ACOS(column_reference or numeric constant)
```

ACOS can only be used with numeric columns or constants.

ASIN Function

The ASIN function returns the arcsine of a number as an angle expressed in radians:

```
ASIN(column_reference or numeric constant)
```

ASIN can only be used with numeric columns or constants.

ATAN Function

The ATAN function returns the arctangent of a number as an angle expressed in radians:

```
ATAN(column_reference or numeric constant)
```

ATAN can only be used with numeric columns or constants.

ATAN2 Function

The ATAN2 function returns the arctangent of x and y coordinates as an angle expressed in radians:

```
ATAN2(column_reference or numeric constant,
              column_reference or numeric constant)
```

ATAN2 can only be used with numeric columns or constants.

CEIL or CEILING Function

The CEIL or CEILING function returns the lowest integer greater than or equal to a number:

```
CEIL(column_reference or numeric constant)
CEILING(column_reference or numeric constant)
```

CEIL or CEILING can only be used with numeric columns or constants.

COS Function

The COS function returns the cosine of an angle:

```
COS(column_reference or numeric constant)
```

COS can only be used with numeric columns or constants.

COT Function

The COT function returns the cotangent of an angle:

```
COT(column_reference or numeric constant)
```

COT can only be used with numeric columns or constants.

DEGREES Function

The DEGREES function converts a number representing radians into degrees:

```
DEGREES(column_reference or numeric constant)
```

DEGREES can only be used with numeric columns or constants.

EXP Function

The EXP function returns the exponential value of a number:

```
EXP(column_reference or numeric constant)
```

EXP can only be used with numeric columns or constants.

FLOOR Function

The FLOOR function returns the highest integer less than or equal to a number:

```
FLOOR(column_reference or numeric constant)
```

FLOOR can only be used with numeric columns or constants.

LOG Function

The LOG function returns the natural logarithm of a number:

```
LOG(column_reference or numeric constant)
```

LOG can only be used with numeric columns or constants.

LOG10 Function

The LOG10 function returns the base 10 logarithm of a number:

```
LOG10(column_reference or numeric constant)
```

LOG10 can only be used with numeric columns or constants.

MOD Function

The MOD function returns the modulus of two integers. The modulus is the remainder that is present when dividing the first integer by the second integer:

```
MOD(column_reference or integer constant,
            column_reference or integer constant)
```

MOD can only be used with integer columns or constants.

PI Function

The PI function returns the ratio of a circle's circumference to its diameter - approximated as 3.1415926535897932385:

```
PI()
```

POWER Function

The POWER function returns value of a base number raised to the specified power:

```
POWER(column_reference or numeric constant
      TO column_reference or numeric constant)
POWER(column_reference or numeric constant,
      column_reference or numeric constant)
```

POWER can only be used with numeric columns or constants.

RADIANS Function

The RADIANS function converts a number representing degrees into radians:

```
RADIANS(column_reference or numeric constant)
```

RADIANS can only be used with numeric columns or constants.

RAND Function

The RAND function returns a random number:

```
RAND([RANGE range of random values])
```

The range value is optional used to limit the random numbers returned to between 0 and the range value specified. If the range is not specified then any number within the full range of numeric values may be returned.

ROUND Function

The ROUND function rounds a numeric value to a specified number of decimal places:

```
ROUND(column_reference or numeric constant
              [TO number of decimal places])
ROUND(column_reference or numeric constant
     [, number of decimal places])
```

The number of decimal places is optional, and if not specified the value returned will be rounded to 0 decimal places.

ROUND can only be used with numeric columns or constants.

Note
The ROUND function performs "normal" rounding where the number is rounded up if the fractional portion beyond the number of decimal places being rounded to is greater than or equal to 5 and down if the fractional portion is less than 5. Also, if using the ROUND function with floating-point values, it is possible to encounter rounding errors due to the nature of floating-point values and their inability to accurately express certain numbers. If you want to eliminate this possibility you should use the CAST function to convert the floating-point column or constant to a BCD value (DECIMAL or NUMERIC data type in SQL). This will allow for the rounding to occur as desired since BCD values can accurately represent these numbers without errors.

SIGN Function

The SIGN function returns -1 if a number is less than 0, 0 if a number is 0, or 1 if a number is greater than 0:

```
SIGN(column_reference or numeric constant)
```

SIGN can only be used with numeric columns or constants.

SIN Function

The SIN function returns the sine of an angle:

```
SIN(column_reference or numeric constant)
```

SIN can only be used with numeric columns or constants.

SQRT Function

The SQRT function returns the square root of a number:

```
SQRT(column_reference or numeric constant)
```

SQRT can only be used with numeric columns or constants.

TAN Function

The TAN function returns the tangent of an angle:

```
TAN(column_reference or numeric constant)
```

TAN can only be used with numeric columns or constants.

TRUNC or TRUNCATE Function

The TRUNC or TRUNCATE function truncates a numeric value to a specified number of decimal places:

```
TRUNC(column_reference or numeric constant
      [TO number of decimal places])
TRUNCATE(column_reference or numeric constant
         [TO number of decimal places])
TRUNC(column_reference or numeric constant
      [, number of decimal places])
TRUNCATE(column_reference or numeric constant
         [, number of decimal places])
```

The number of decimal places is optional, and if not specified the value returned will be truncated to 0 decimal places.

TRUNC or TRUNCATE can only be used with numeric columns or constants.

Note
If using the TRUNC or TRUNCATE function with floating-point values, it is possible to encounter truncation errors due to the nature of floating-point values and their inability to accurately express certain numbers. If you want to eliminate this possibility you should use the CAST function to convert the floating-point column or constant to a BCD value (DECIMAL or NUMERIC data type in SQL). This will allow for the truncation to occur as desired since BCD values can accurately represent these numbers without errors.

Boolean Functions

Use boolean functions to manipulate any values in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following boolean functions:

Function	Description
IF	Performs IF..ELSE type of inline expression handling.
IFNULL	Performs IF..ELSE type of inline expression handling specifically for NULL values.
NULLIF	Returns a NULL if two values are equivalent.
COALESCE	Returns the first non-NULL value from a list of expressions.

IF Function

The IF function performs inline IF..ELSE boolean expression handling:

```
IF(boolean expression THEN result expression
   ELSE result expression)
```

```
IF(boolean expression, result expression,
   result expression)
```

Both result expressions must be of the same resultant data type. Use the CAST function to ensure that both expressions are of the same data type.

In the following example, if the Category column contains the value 'WRASSE', then the column value returned will be the Common_Name column, otherwise it will be the Species Name column:

```
SELECT IF(Upper(Category)='WRASSE'
THEN Common_Name
ELSE "Species Name") AS Name
FROM Biolife
```

The IF function can be used in WHERE clause comparisons to cause a conditional comparison:

```
SELECT *
FROM Employee
WHERE IF(LastName='Young' THEN PhoneExt='233' ELSE PhoneExt='22')
```

IFNULL Function

The IFNULL function performs inline IF..ELSE boolean expression handling specifically on NULL values:

```
IFNULL(expression THEN result expression
       ELSE result expression)
IFNULL(expression, result expression,
       result expression)
```

Both result expressions must be of the same resultant data type. Use the CAST function to ensure that both expressions are of the same data type.

In the following example, if the Category column contains a NULL value, then the column value returned will be the Common_Name column, otherwise it will be the Species Name column:

```
SELECT IFNULL(Category THEN Common_Name
ELSE "Species Name") AS Name
FROM Biolife
```

The IFNULL function can be used in WHERE clause comparisons to cause a conditional comparison:

```
SELECT *
FROM Employee
WHERE IFNULL(Salary THEN 10000 ELSE Salary) > 8000
```

NULLIF Function

The NULLIF function returns a NULL if the two values passed as parameters are equal:

```
NULLIF(expression,expression)
```

Both expressions must be of the same data type. Use the CAST function to ensure that both expressions are of the same data type.

In the following example, if the EmpNo column contains the value 14, then the value returned will be NULL, otherwise it will be the EmpNo column value:

```
SELECT NULLIF(EmpNo,14) AS EmpNo
FROM Orders
```

The NULLIF function can be used in WHERE clause comparisons to cause a conditional comparison:

```
SELECT *
FROM Employee
WHERE NULLIF(Salary,10000) > 8000
```

COALESCE Function

The COALESCE function returns the first non-NULL value from a list of expressions:

```
COALESCE(expression [, expression [, expression]])
```

All expressions must be of the same resultant data type. Use the CAST function to ensure that all expressions are of the same data type.

In the following example, if the Category column contains a NULL value, then the column value returned will be the Common_Name column. If the Common_name column contains a NULL, then the literal string 'No Name' will be returned:

```
SELECT COALESCE(Category,Common_Name,'No Name') AS Name
FROM Biolife
```

Aggregate Functions

Use aggregate functions to perform aggregate calculations on values in SELECT queries containing a GROUP BY clause. DBISAM's SQL supports the following aggregate functions:

Function	Description
AVG	Averages all numeric values in a column.
COUNT	Counts the total number of rows or the number of rows where the specified column is not NULL.
MAX	Determines the maximum value in a column.
MIN	Determines the minimum value in a column.
STDDEV	Calculates the standard deviation of all numeric values in a column.
SUM	Totals all numeric values in a column.
RUNSUM	Totals all numeric values in a column in a running total.
LIST	Concatenates all string values in a column using a delimeter

AVG Function

The AVG function returns the average of the values in a specified column or expression. The syntax is as follows:

```
AVG(column_reference or expression)
```

Use AVG to calculate the average value for a numeric column. As an aggregate function, AVG performs its calculation aggregating values in the same column(s) across all rows in a dataset. The dataset may be the entire table, a filtered dataset, or a logical group produced by a GROUP BY clause. Column values of zero are included in the averaging, so values of 1, 2, 3, 0, 0, and 0 result in an average of 1. NULL column values are not counted in the calculation. The following is an example of using the AVG function to calculate the average order amount for all orders:

```
SELECT AVG(ItemsTotal)
FROM Orders
```

AVG returns the average of values in a column or the average of a calculation using a column performed for each row (a calculated field). The following example shows how to use the AVG function to calculate an average order amount and tax amount for all orders:

```
SELECT AVG(ItemsTotal) AS AverageTotal,
AVG(ItemsTotal * 0.0825) AS AverageTax
FROM Orders
```

When used with a GROUP BY clause, AVG calculates one value for each group. This value is the aggregation of the specified column for all rows in each group. The following example aggregates the average value for the ItemsTotal column in the Orders table, producing a subtotal for each company in the Customer table:

```
SELECT c."Company",
AVG(o."ItemsTotal") AS Average,
MAX(o."ItemsTotal") AS Biggest,
MIN(o."ItemsTotal") AS Smallest
FROM "Customer.dat" c, "Orders.dat" o
WHERE (c."CustNo" = o."CustNo")
GROUP BY c."Company"
ORDER BY c."Company"
```

AVG operates only on numeric values.

COUNT Function

The COUNT function returns the number of rows that satisfy a query's search condition or the number of rows where the specified column is not NULL. The syntax is as follows:

```
COUNT(* | column_reference or expression)
```

Use COUNT to count the number of rows retrieved by a SELECT statement. The SELECT statement may be a single-table or multi-table query. The value returned by COUNT reflects a reduced row count produced by a filtered dataset. The following example returns the total number of rows in the Averaging source table with a non-NULL Amount column:

```
SELECT COUNT(Amount)
FROM Averaging
```

The following example returns the total number of rows in the filtered Orders source table irrespective of any NULL column values:

```
SELECT COUNT(*)
FROM Orders
WHERE (Orders.ItemsTotal > 5000)
```

MAX Function

The MAX function returns the largest value in the specified column. The syntax is as follows:

```
MAX(column_reference or expression)
```

Use MAX to calculate the largest value for a string, numeric, date, time, or timestamp column. As an aggregate function, MAX performs its calculation aggregating values in the same column(s) across all rows in a dataset. The dataset may be the entire table, a filtered dataset, or a logical group produced by a GROUP BY clause. Column values of zero are included in the aggregation. NULL column values are not counted in the calculation. If the number of qualifying rows is zero, MAX returns a NULL value. The following is an example of using the MAX function to calculate the largest order amount for all orders:

```
SELECT MAX(ItemsTotal)
FROM Orders
```

MAX returns the largest value in a column or a calculation using a column performed for each row (a calculated field). The following example shows how to use the MAX function to calculate the largest order amount and tax amount for all orders:

```
SELECT MAX(ItemsTotal) AS HighestTotal,
MAX(ItemsTotal * 0.0825) AS HighestTax
FROM Orders
```

When used with a GROUP BY clause, MAX returns one calculation value for each group. This value is the aggregation of the specified column for all rows in each group. The following example aggregates the largest value for the ItemsTotal column in the Orders table, producing a subtotal for each company in the Customer table:

```
SELECT c."Company",
AVG(o."ItemsTotal") AS Average,
MAX(o."ItemsTotal") AS Biggest,
MIN(o."ItemsTotal") AS Smallest
FROM "Customer.dat" c, "Orders.dat" o
WHERE (c."CustNo" = o."CustNo")
GROUP BY c."Company"
ORDER BY c."Company"
```

MAX can be used with all string, numeric, date, time, and timestamp columns. The return value is of the same type as the column.

MIN Function

The MIN function returns the smallest value in the specified column. The syntax is as follows:

```
MIN(column_reference or expression)
```

Use MIN to calculate the smallest value for a string, numeric, date, time, or timestamp column. As an aggregate function, MAX performs its calculation aggregating values in the same column(s) across all rows in a dataset. The dataset may be the entire table, a filtered dataset, or a logical group produced by a GROUP BY clause. Column values of zero are included in the aggregation. NULL column values are not counted in the calculation. If the number of qualifying rows is zero, MAX returns a NULL value. The following is an example of using the MAX function to calculate the smallest order amount for all orders:

```
SELECT MIN(ItemsTotal)
FROM Orders
```

MIN returns the smallest value in a column or a calculation using a column performed for each row (a calculated field). The following example shows how to use the MIN function to calculate the smallest order amount and tax amount for all orders:

```
SELECT MIN(ItemsTotal) AS LowestTotal,
MIN(ItemsTotal * 0.0825) AS LowestTax
FROM Orders
```

When used with a GROUP BY clause, MIN returns one calculation value for each group. This value is the aggregation of the specified column for all rows in each group. The following example aggregates the smallest value for the ItemsTotal column in the Orders table, producing a subtotal for each company in the Customer table:

```
SELECT c."Company",
AVG(o."ItemsTotal") AS Average,
MAX(o."ItemsTotal") AS Biggest,
MIN(o."ItemsTotal") AS Smallest
FROM "Customer.dat" c, "Orders.dat" o
WHERE (c."CustNo" = o."CustNo")
GROUP BY c."Company"
ORDER BY c."Company"
```

MIN can be used with all string, numeric, date, time, and timestamp columns. The return value is of the same type as the column.

STDDEV Function

The STDDEV function returns the standard deviation of the values in a specified column or expression. The syntax is as follows:

```
STDDEV(column_reference or expression)
```

Use STDDEV to calculate the standard deviation value for a numeric column. As an aggregate function, STDDEV performs its calculation aggregating values in the same column(s) across all rows in a dataset. The dataset may be the entire table, a filtered dataset, or a logical group produced by a GROUP BY clause. NULL column values are not counted in the calculation. The following is an example of using the STDDEV function to calculate the standard deviation for a set of test scores:

```
SELECT STDDEV(TestScore)
FROM Scores
```

When used with a GROUP BY clause, STDDEV calculates one value for each group. This value is the aggregation of the specified column for all rows in each group.

STDDEV operates only on numeric values.

SUM Function

The SUM function calculates the sum of values for a column. The syntax is as follows:

```
SUM(column_reference or expression)
```

Use SUM to sum all the values in the specified column. As an aggregate function, SUM performs its calculation aggregating values in the same column(s) across all rows in a dataset. The dataset may be the entire table, a filtered dataset, or a logical group produced by a GROUP BY clause. Column values of zero are included in the aggregation. NULL column values are not counted in the calculation. If the number of qualifying rows is zero, SUM returns a NULL value. The following is an example of using the SUM function to calculate the total order amount for all orders:

```
SELECT SUM(ItemsTotal)
FROM Orders
```

SUM returns the total sum of a column or a calculation using a column performed for each row (a calculated field). The following example shows how to use the SUM function to calculate the total order amount and tax amount for all orders:

```
SELECT SUM(ItemsTotal) AS Total,
SUM(ItemsTotal * 0.0825) AS TotalTax
FROM orders
```

When used with a GROUP BY clause, SUM returns one calculation value for each group. This value is the aggregation of the specified column for all rows in each group. The following example aggregates the total value for the ItemsTotal column in the Orders table, producing a subtotal for each company in the Customer table:

```
SELECT c."Company",
SUM(o."ItemsTotal") AS SubTotal
FROM "Customer.dat" c, "Orders.dat" o
WHERE (c."CustNo" = o."CustNo")
GROUP BY c."Company"
ORDER BY c."Company"
```

SUM operates only on numeric values.

RUNSUM Function

The RUNSUM function calculates the sum of values for a column in a running total. The syntax is as follows:

```
RUNSUM(column_reference or expression)
```

Use RUNSUM to sum all the values in the specified column in a continuous running total. The RUNSUM function is identical to the SUM function except for the fact that it does not reset itself when sub-totalling.

> **Note**
> The running total is only calculated according to the implicit order of the GROUP BY fields and is not affected by an ORDER BY statement.

LIST Function

The LIST function calculates the concatenation of string values for a column, using a delimiter to separate each value. The syntax is as follows:

```
LIST(column_reference or expression[,delimiter])
```

Use LIST to concatenate all the string values in the specified column into a single string value, using a delimiter to separate one value from the next. If the delimiter is not specified, then the default delimiter is the comma (,).

AutoInc Functions

Use autoinc functions to return the last autoinc value from a given table in INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following autoinc functions:

Function	Description
LASTAUTOINC	Returns the last autoinc value from a specified table.
IDENT_CURRENT	Same as LASTAUTOINC, with a different name.

LASTAUTOINC Function

The LASTAUTOINC function returns the last autoinc value from a specified table. The syntax is as follows:

```
LASTAUTOINC(table name constant)
```

The LASTAUTOINC function will return the last autoinc value from the specified table relative to the start of the SQL statement currently referencing the LASTAUTOINC function. Because of this, it is possible for LASTAUTOINC to not return the most recent last autoinc value for the specified table. It is usually recommended that you only use this function within the scope of a transaction in order to guarantee that you have retrieved the correct last autoinc value from the table. The following example illustrates how this would be accomplished using an SQL script and a master-detail insert:

```
START TRANSACTION;

INSERT INTO customer (company) VALUES ('Test');

INSERT INTO orders (custno,empno) VALUES (LASTAUTOINC('customer'),100);
```

```
INSERT INTO orders (custno,empno) VALUES (LASTAUTOINC('customer'),200);

COMMIT FLUSH;
```

Full Text Indexing Functions

Use full text indexing functions to search for specific words in a given column in SELECT, INSERT, UPDATE, or DELETE queries. The word search is controlled by the text indexing parameters for the table in which the column resides. DBISAM's SQL supports the following word search functions:

Function	Description
TEXTSEARCH	Performs an optimized text word search on a field, if the field is part of the full text index for the table, or a brute-force text word search if not.
TEXTOCCURS	Counts the number of times a list of words appears in a field based upon the full text indexing parameters for the table.

TEXTSEARCH Function

The TEXTSEARCH function searches a column for a given set of words in a search string constant. The syntax is as follows:

```
TEXTSEARCH(search string constant
           IN column_reference)
TEXTSEARCH(search string constant,
           column_reference)
```

The optimization of the TEXTSEARCH function is controlled by whether the column being searched is part of the full text index for the table in which the column resides. If the column is not part of the full text index then the search will resort to a brute-force scan of the contents of the column in every record that satisifies any prior conditions in the WHERE clause. Also, the parsing of the list of words in the search string constant is controlled by the text indexing parameters for the table in which the column being searched resides. Please see the Full Text Indexing topic for more information.

In the following example, the words 'DATABASE QUERY SPEED' are searched for in the TextBody column:

```
SELECT GroupNo, No
FROM article
WHERE TEXTSEARCH('DATABASE QUERY SPEED' IN TextBody)
```

TEXTSEARCH returns a boolean value indicating whether the list of words exists in the column for a given record. TEXTSEARCH can only be used with string or memo columns.

TEXTOCCURS Function

The TEXTOCCURS function searches a column for a given set of words in a search string constant and returns the number of times the words occur in the column. The syntax is as follows:

```
TEXTOCCURS(search string constant
           IN column_reference)
TEXTOCCURS(search string constant,
           column_reference)
```

TEXTOCCURS is always a brute-force operation and accesses the actual column contents to perform its functionality, unlike the TEXTSEARCH function which can be optimized by adding the column being searched to the full text index for the table. Also, the parsing of the list of words in the search string constant is controlled by the text indexing parameters for the table in which the column being searched resides. Please see the Full Text Indexing topic for more information.

In the following example, the number of occurrences of the words 'DATABASE QUERY SPEED' in the TextBody column are used to order the results of a TEXTSEARCH query in order to provide ranking for the text search:

```
SELECT GroupNo, No,
TEXTOCCURS('DATABASE QUERY SPEED' IN TextBody) AS NumOccurs
FROM article
WHERE TEXTSEARCH('DATABASE QUERY SPEED' IN TextBody)
ORDER BY 3 DESC
```

TEXTOCCURS returns an integer value indicating the total number of times the list of words occurs in the column for a given record. TEXTOCCURS can only be used with string or memo columns.

Data Conversion Functions

Use data conversion functions to convert values from one type to another in SELECT, INSERT, UPDATE, or DELETE queries. DBISAM's SQL supports the following data conversion functions:

Function	Description
EXTRACT	Extracts the year, month, week, day of week, or day value of a date or the hours, minutes, or seconds value of a time.
CAST	Converts a given data value from one data type to another.
YEARSFROMMSECS	Takes milliseconds and returns the number of years.
DAYSFROMMSECS	Takes milliseconds and returns the number of days (as a remainder of the above years, not as an absolute).
HOURSFROMMSECS	Takes milliseconds and returns the number of hours (as a remainder of the above years and days, not as an absolute).
MINSFROMMSECS	Takes milliseconds and returns the number of minutes (as a remainder of the above years, days, and hours, not as an absolute).
SECSFROMMSECS	Takes milliseconds and returns the number of seconds (as a remainder of the above years, days, hours, and minutes, not as an absolute).

MSECSFROMMSECS	Takes milliseconds and returns the number of milliseconds (as a remainder of the above years, days, hours, minutes, and seconds, not as an absolute).

EXTRACT Function

The EXTRACT function returns a specific value from a date, time, or timestamp value. The syntax is as follows:

```
EXTRACT(extract_value
        FROM column_reference or expression)
EXTRACT(extract_value,
        column_reference or expression)
```

Use EXTRACT to return the year, month, week, day of week, day, hours, minutes, seconds, or milliseconds from a date, time, or timestamp column. EXTRACT returns the value for the specified element as an integer.

The extract_value parameter may contain any one of the specifiers:

```
YEAR
MONTH
WEEK
DAYOFWEEK
DAYOFYEAR
DAY
HOUR
MINUTE
SECOND
MSECOND
```

The specifiers YEAR, MONTH, WEEK, DAYOFWEEK, DAYOFYEAR, and DAY can only be used with date and timestamp columns. The following example shows how to use the EXTRACT function to display the various elements of the SaleDate column:

```
SELECT SaleDate,
EXTRACT(YEAR FROM SaleDate) AS YearNo,
EXTRACT(MONTH FROM SaleDate) AS MonthNo,
EXTRACT(WEEK FROM SaleDate) AS WeekNo,
EXTRACT(DAYOFWEEK FROM SaleDate) AS WeekDayNo,
EXTRACT(DAYOFYEAR FROM SaleDate) AS YearDayNo,
EXTRACT(DAY FROM SaleDate) AS DayNo
FROM Orders
```

The following example uses a DOB column (containing birthdates) to filter those rows where the date is in the month of May. The month field from the DOB column is retrieved using the EXTRACT function and compared to 5, May being the fifth month:

```
SELECT DOB, LastName, FirstName
FROM People
WHERE (EXTRACT(MONTH FROM DOB) = 5)
```

Note
The WEEK and DAYOFWEEK parameters will return the week number and the day of the week according to ANSI/ISO standards. This means that the first week of the year (week 1) is the first week that contains the first Thursday in January and January 4th and the first day of the week (day 1) is Monday. Also, while ANSI-standard SQL provides the EXTRACT function specifiers TIMEZONE_HOUR and TIMEZONE_MINUTE, these specifiers are not supported in DBISAM's SQL.

EXTRACT operates only on date, time, and timestamp values.

CAST Function

The CAST function converts a specified value to the specified data type. The syntax is as follows:

```
CAST(column_reference AS data_type)
CAST(column_reference,data_type)
```

Use CAST to convert the value in the specified column to the data type specified. CAST can also be applied to literal and calculated values. CAST can be used in the columns list of a SELECT statement, in the predicate for a WHERE clause, or to modify the update atom of an UPDATE statement.

The data type parameter may be any valid SQL data type that is a valid as a destination type for the source data being converted. Please see the Data Types and NULL Support topic for more information.

The statement below converts a timestamp column value to a date column value:

```
SELECT CAST(SaleDate AS DATE)
FROM ORDERS
```

Converting a column value with CAST allows use of other functions or predicates on an otherwise incompatible data type, such as using the SUBSTRING function on a date column:

```
SELECT SaleDate,
SUBSTRING(CAST(CAST(SaleDate AS DATE) AS CHAR(10)) FROM 1 FOR 1)
FROM Orders
```

Note
All conversions of dates or timestamps to strings are done using the 24-hour clock (military time).

YEARSFROMMSECS Function

The YEARSFROMMSECS function takes milliseconds and returns the number of years. The syntax is as follows:

```
YEARSFROMMSECS(column_reference or expression)
```

Use YEARSFROMMSECS to return the number of years present in a milliseconds value as an integer value.

DAYSFROMMSECS Function

The DAYSFROMMSECS function takes milliseconds and returns the number of days as a remainder of the number of years present in the milliseconds. The syntax is as follows:

```
DAYSFROMMSECS(column_reference or expression)
```

Use DAYSFROMMSECS to return the number of days present in a milliseconds value as an integer value. The number of days is represented as the remainder of days once the number of years is removed from the milliseconds value using the YEARSFROMMSECS function.

HOURSFROMMSECS Function

The HOURSFROMMSECS function takes milliseconds and returns the number of hours as a remainder of the number of years and days present in the milliseconds. The syntax is as follows:

```
HOURSFROMMSECS(column_reference or expression)
```

Use HOURSFROMMSECS to return the number of hours present in a milliseconds value as an integer value. The number of hours is represented as the remainder of hours once the number of years and days is removed from the milliseconds value using the YEARSFROMMSECS and DAYSFROMMSECS functions.

MINSFROMMSECS Function

The MINSFROMMSECS function takes milliseconds and returns the number of minutes as a remainder of the number of years, days, and hours present in the milliseconds. The syntax is as follows:

```
MINSFROMMSECS(column_reference or expression)
```

Use MINSFROMMSECS to return the number of minutes present in a milliseconds value as an integer value. The number of minutes is represented as the remainder of minutes once the number of years, days, and hours is removed from the milliseconds value using the YEARSFROMMSECS, DAYSFROMMSECS, and HOURSFROMMSECS functions.

SECSFROMMSECS Function

The SECSFROMMSECS function takes milliseconds and returns the number of seconds as a remainder of

the number of years, days, hours, and minutes present in the milliseconds. The syntax is as follows:

```
SECSFROMMSECS(column_reference or expression)
```

Use SECSFROMMSECS to return the number of seconds present in a milliseconds value as an integer value. The number of seconds is represented as the remainder of seconds once the number of years, days, hours, and minutes is removed from the milliseconds value using the YEARSFROMMSECS, DAYSFROMMSECS, HOURSFROMMSECS, and MINSFROMMSECS functions.

MSECSFROMMSECS Function

The MSECSFROMMSECS function takes milliseconds and returns the number of milliseconds as a remainder of the number of years, days, hours, minutes, and seconds present in the milliseconds. The syntax is as follows:

```
MSECSFROMMSECS(column_reference or expression)
```

Use MSECSFROMMSECS to return the number of milliseconds present in a milliseconds value as an integer value. The number of milliseconds is represented as the remainder of milliseconds once the number of years, days, hours, minutes, and seconds is removed from the milliseconds value using the YEARSFROMMSECS, DAYSFROMMSECS, HOURSFROMMSECS, MINSFROMMSECS, and SECSFROMMSECS functions.

4.7 SELECT Statement

Introduction

The SQL SELECT statement is used to retrieve data from tables. You can use the SELECT statement to:

- Retrieve a single row, or part of a row, from a table, referred to as a singleton select.
- Retrieve multiple rows, or parts of rows, from a table.
- Retrieve related rows, or parts of rows, from a join of two or more tables.

Syntax

```
SELECT [DISTINCT | ALL] * | column
[AS correlation_name | correlation_name], [column...]

[INTO destination_table]

FROM table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE]

[[[[INNER | [LEFT | RIGHT] OUTER JOIN] table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE]
ON join_condition]

[WHERE predicates]

[GROUP BY group_list]

[HAVING predicates]

[[UNION | EXCEPT| INTERSECT] [ALL] [SELECT...]]

[ORDER BY order_list [NOCASE]]

[TOP number_of_rows]

[LOCALE locale_name | LOCALE CODE locale_code]

[ENCRYPTED WITH password]

[NOJOINOPTIMIZE]
[JOINOPTIMIZECOSTS]
[NOWHEREJOINS]
```

The SELECT clause defines the list of items returned by the SELECT statement. The SELECT clause uses a comma-separated list composed of: table columns, literal values, and column or literal values modified by functions. You cannot use parameters in this list of items. Use an asterisk to retrieve values from all columns. Columns in the column list for the SELECT clause may come from more than one table, but can only come from those tables listed in the FROM clause. The FROM clause identifies the table(s) from which data is retrieved.

The following example retrieves data for two columns in all rows of a table:

```
SELECT CustNo, Company
FROM Orders
```

You can use the AS keyword to specify a column correlation name, or alternately you can simply just specify the column correlation name after the selected column. The following example uses both methods to give each selected column a more descriptive name in the query result set:

```
SELECT Customer.CustNo AS "Customer #",
Customer.Company AS "Company Name",
Orders.OrderNo "Order #",
SUM(Items.Qty) "Total Qty"
FROM Customer LEFT OUTER JOIN Orders ON Customer.Custno=Orders.Custno
LEFT OUTER JOIN Items ON Orders.OrderNo=Items.OrderNo
WHERE Customer.Company LIKE '%Diver%'
GROUP BY 1,2
ORDER BY 1
```

Use DISTINCT to limit the retrieved data to only distinct rows. The distinctness of rows is based on the combination of all of the columns in the SELECT clause columns list. DISTINCT can only be used with simple column types like string and integer; it cannot be used with complex column types like blob.

INTO Clause

The INTO clause specifies a table into which the query results are generated. The syntax is as follows:

```
INTO destination_table
```

Use an INTO clause to specify the table where the query results will be stored when the query has completed execution. The following example shows how to generate all of the orders in the month of January as a table on disk named "Results":

```
SELECT *
INTO "Results"
FROM "Orders"
```

If you do not specify a drive and directory in the destination table name, for local sessions, or a database name in the destination table name, for remote sessions, then the destination table will be created in the current active database for the query being executed.

The following examples show the different options for the INTO clause and their resultant destination table names.

This example produces a destination table in the current database called "Results":

```
SELECT *
INTO "Results"
FROM "Orders"
```

This example produces a destination table called "Results" in the specified local database directory (valid for local sessions only):

```
SELECT *
INTO "c:\MyData\Results"
FROM "Orders"
```

This example produces a destination table called "Results" in the specified database (valid for remote sessions only):

```
SELECT *
INTO "\MyRemoteDB\Results"
FROM "Orders"
```

This example produces an in-memory destination table called "Results":

```
SELECT *
INTO "\Memory\Results"
FROM "Orders"
```

There are some important caveats when using the INTO clause:

- The INTO clause creates the resultant table from scratch, so if a table with the same name in the same location already exists, it will be overwritten. This also means that any indexes defined for the table will be removed or modified, even if the result set columns match those of the existing table.
- You must make sure that you close the query before trying to access the destination table with another table component. If you do not an exception will be raised.
- You must make sure to delete the table after you are done if you don't wish to leave it on disk or in-memory for further use.
- Remote sessions can only produce tables that are accessible from the database server and cannot automatically create a local table from a query on the database server by specifying a local path for the INTO clause. The path for the INTO clause must be accessible from the database server in order for the query to be successfully executed.
- The destination table cannot be passed to the INTO clause via a parameter.

FROM Clause

The FROM clause specifies the tables from which a SELECT statement retrieves data. The syntax is as follows:

```
FROM table_reference [AS] [correlation_name]
[, table_reference...]
```

Use a FROM clause to specify the table or tables from which a SELECT statement retrieves data. The value for a FROM clause is a comma-separated list of table names. Specified table names must follow DBISAM's SQL naming conventions for tables. Please see the Naming Conventions topic for more information. The following SELECT statement below retrieves data from a single table:

```
SELECT *
FROM "Customer"
```

The following SELECT statement below retrieves data from a single in-memory table:

```
SELECT *
FROM "\Memory\Customer"
```

You can use the AS keyword to specify a table correlation name, or alternately you can simply just specify the table correlation name after the source table name. The following example uses both methods to give each source table a shorter name to be used in qualifying source columns in the query:

```
SELECT c.CustNo AS "Customer #",
c.Company AS "Company Name",
o.OrderNo "Order #",
SUM(i.Qty) "Total Qty"
FROM Customer AS c LEFT OUTER JOIN Orders AS o ON c.Custno=o.Custno
LEFT OUTER JOIN Items i ON o.OrderNo=i.OrderNo
WHERE c.Company LIKE '%Diver%'
GROUP BY 1,2
ORDER BY 1
```

Use the EXCLUSIVE keyword to specify that the table should be opened exclusively.

Note
Be careful when using the EXCLUSIVE keyword with a table that is specified more than once in the same query, as is the case with recursive relationships between a table and itself.

See the section below entitled JOIN clauses for more information on retrieving data from multiple tables in a single SELECT query.

The table reference cannot be passed to a FROM clause via a parameter.

JOIN Clauses

There are three types of JOIN clauses that can be used in the FROM clause to perform relational joins between source tables. The implicit join condition is always Cartesian for source tables without an explicit JOIN clause.

Join Type	Description

Cartesian	Joins two tables, matching each row of one table with each row from the other.
INNER	Joins two tables, filtering out non-matching rows.
OUTER	Joins two tables, retaining non-matching rows.

Cartesan Join

A Cartesian join connects two tables in a non-relational manner. The syntax is as follows:

```
FROM table_reference, table_reference [,table_reference...]
```

Use a Cartesian join to connect the column of two tables into one result set, but without correlation between the rows from the tables. Cartesian joins match each row of the source table with each row of the joining table. No column comparisons are used, just simple association. If the source table has 10 rows and the joining table has 10, the result set will contain 100 rows as each row from the source table is joined with each row from the joined table.

INNER JOIN Clause

An INNER join connects two tables based on column values common between the two, excluding non-matches. The syntax is as follows:

```
FROM table_reference
[INNER] JOIN table_reference ON predicate
[[INNER] JOIN table_reference ON predicate...]
```

Use an INNER JOIN to connect two tables, a source and joining table, that have values from one or more columns in common. One or more columns from each table are compared in the ON clause for equal values. For rows in the source table that have a match in the joining table, the data for the source table rows and matching joining table rows are included in the result set. Rows in the source table without matches in the joining table are excluded from the joined result set. In the following example the Customer and Orders tables are joined based on values in the CustNo column, which each table contains:

```
SELECT *
FROM Customer c INNER JOIN Orders o ON (c.CustNo=o.CustNo)
```

More than one table may be joined with an INNER JOIN. One use of the INNER JOIN operator and corresponding ON clause is required for each each set of two tables joined. One columns comparison predicate in an ON clause is required for each column compared to join each two tables. The following example joins the Customer table to Orders, and then Orders to Items. In this case, the joining table Orders acts as a source table for the joining table Items:

```
SELECT *
FROM Customer c JOIN Orders o ON (c.CustNo = o.CustNo)
JOIN Items i ON (o.OrderNo = i.OrderNo)
```

Tables may also be joined using a concatenation of multiple column values to produce a single value for the join comparison predicate. In the following example the ID1 and ID2 columns in the Joining table are concatenated and compared with the values in the single column ID in Source:

```
SELECT *
FROM Source s INNER JOIN Joining j ON (s.ID = j.ID1 || j.ID2)
```

OUTER JOIN Clause

The OUTER JOIN clause connects two tables based on column values common between the two, including non-matches. The syntax is as follows:

```
FROM table_reference LEFT | RIGHT [OUTER]
JOIN table_reference ON predicate
[LEFT | RIGHT [OUTER] JOIN table_reference ON predicate...]
```

Use an OUTER JOIN to connect two tables, a source and joining table, that have one or more columns in common. One or more columns from each table are compared in the ON clause for equal values. The primary difference between inner and outer joins is that, in outer joins rows from the source table that do not have a match in the joining table are not excluded from the result set. Columns from the joining table for rows in the source table without matches have NULL values.

In the following example the Customer and Orders tables are joined based on values in the CustNo column, which each table contains. For rows from Customer that do not have a matching value between Customer.CustNo and Orders.CustNo, the columns from Orders contain NULL values:

```
SELECT *
FROM Customer c LEFT OUTER JOIN Orders o ON (c.CustNo = o.CustNo)
```

The LEFT modifier causes all rows from the table on the left of the OUTER JOIN operator to be included in the result set, with or without matches in the table to the right. If there is no matching row from the table on the right, its columns contain NULL values. The RIGHT modifier causes all rows from the table on the right of the OUTER JOIN operator to be included in the result set, with or without matches. If there is no matching row from the table on the left, its columns contain NULL values.

More than one table may be joined with an OUTER JOIN. One use of the OUTER JOIN operator and corresponding ON clause is required for each each set of two tables joined. One column comparison predicate in an ON clause is required for each column compared to join each two tables. The following example joins the Customer table to the Orders table, and then Orders to Items. In this case, the joining table Orders acts as a source table for the joining table Items:

```
SELECT *
FROM Customer c LEFT OUTER JOIN Orders o ON (c.CustNo = o.CustNo)
LEFT OUTER JOIN Items i ON (o.OrderNo = i.OrderNo)
```

Tables may also be joined using expressions to produce a single value for the join comparison predicate.

In the following example the ID1 and ID2 columns in Joining are separately compared with two values produced by the SUBSTRING function using the single column ID in Source:

```
SELECT *
FROM Source s RIGHT OUTER JOIN Joining j
ON (SUBSTRING(s.ID FROM 1 FOR 2) = j.ID1) AND
(SUBSTRING(s.ID FROM 3 FOR 1) = j.ID2)
```

WHERE Clause

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

```
WHERE predicates
```

Use a WHERE clause to limit the effect of a SELECT statement to a subset of rows in the table, and the clause is optional.

The value for a WHERE clause is one or more logical expressions, or predicates, that evaluate to true or false for each row in the table. Only those rows where the predicates evaluate to TRUE are retrieved by the SELECT statement. For example, the SELECT statement below retrieves all rows where the State column contains a value of 'CA':

```
SELECT Company, State
FROM Customer
WHERE State='CA'
```

A column used in the WHERE clause of a statement is not required to also appear in the SELECT clause of that statement. In the preceding statement, the State column could be used in the WHERE clause even if it was not also in the SELECT clause.

Multiple predicates must be separated by one of the logical operators OR or AND. Each predicate can be negated with the NOT operator. Parentheses can be used to isolate logical comparisons and groups of comparisons to produce different row evaluation criteria. For example, the SELECT statement below retrieves all rows where the State column contains a value of 'CA' or a value of 'HI':

```
SELECT Company, State
FROM Customer
WHERE (State='CA') OR (State='HI')
```

Subqueries are supported in the WHERE clause. A subquery works like a search condition to restrict the number of rows returned by the outer, or "parent" query. Such subqueries must be valid SELECT statements. SELECT subqueries cannot be correlated in DBISAM's SQL, i.e. they cannot refer to columns in the outer (or "parent") statement. In the following statement, the subquery is said to be un-correlated:

```
SELECT *
FROM "Clients" C
```

```
WHERE C.Acct_Nbr IN
  (SELECT H.Acct_Nbr
  FROM "Holdings" H
  WHERE H.Pur_Date BETWEEN '1994-01-01' AND '1994-12-31')
```

Note
Column correlation names cannot be used in filter comparisons in the WHERE clause. Use the actual column name instead.

A WHERE clause filters data prior to the aggregation of a GROUP BY clause. For filtering based on aggregated values, use a HAVING clause.

Columns devoid of data contain NULL values. To filter using such column values, use the IS NULL predicate.

GROUP BY Clause

The GROUP BY clause combines rows with column values in common into single rows for the SELECT statement. The syntax is as follows:

```
GROUP BY column_reference [, column reference...]
```

Use a GROUP BY clause to cause an aggregation process to be repeated once for each group of similar rows. Similarity between rows is determined by the distinct values (or combination of values) in the columns specified in the GROUP BY. For instance, a query with a SUM function produces a result set with a single row with the total of all the values for the column used in the SUM function. But when a GROUP BY clause is added, the SUM function performs its summing action once for each group of rows. In statements that support a GROUP BY clause, the use of a GROUP BY clause is optional. A GROUP BY clause becomes necessary when both aggregated and non-aggregated columns are included in the same SELECT statement.

In the statement below, the SUM function produces one subtotal of the ItemsTotal column for each distinct value in the CustNo column (i.e., one subtotal for each different customer):

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

The value for the GROUP BY clause is a comma-separated list of columns. Each column in this list must meet the following criteria:

- Be in one of the tables specified in the FROM clause of the query.
- Also be in the SELECT clause of the query.
- Cannot have an aggregate function applied to it (in the SELECT clause).
- Cannot be a BLOB column.

When a GROUP BY clause is used, all table columns in the SELECT clause of the query must meet at least one of the following criteria, or it cannot be included in the SELECT clause:

- Be in the GROUP BY clause of the query.
- Be the subject of an aggregate function.

Literal values in the SELECT clause are not subject to the preceding criteria and are not required to be in the GROUP BY clause in addition to the SELECT clause.

The distinctness of rows is based on the columns in the column list specified. All rows with the same values in these columns are combined into a single row (or logical group). Columns that are the subject of an aggregate function have their values across all rows in the group combined. All columns not the subject of an aggregate function retain their value and serve to distinctly identify the group. For example, in the SELECT statement below, the values in the Sales column are aggregated (totalled) into groups based on distinct values in the Company column. This produces total sales for each company:

```
SELECT C.Company, SUM(O.ItemsTotal) AS TotalSales
FROM Customer C, Orders O
WHERE C.CustNo=O.CustNo
GROUP BY C.Company
ORDER BY C.Company
```

A column may be referenced in a GROUP BY clause by a column correlation name, instead of actual column names. The statement below forms groups using the first column, Company, represented by the column correlation name Co:

```
SELECT C.Company Co, SUM(O.ItemsTotal) AS TotalSales
FROM Customer C, Orders O
WHERE C.CustNo=O.CustNo
GROUP BY Co
ORDER BY 1
```

HAVING Clause

The HAVING clause specifies filtering conditions for a SELECT statement. The syntax is as follows:

```
HAVING predicates
```

Use a HAVING clause to limit the rows retrieved by a SELECT statement to a subset of rows where aggregated column values meet the specified criteria. A HAVING clause can only be used in a SELECT statement when:

- The statement also has a GROUP BY clause.
- One or more columns are the subjects of aggregate functions.

The value for a HAVING clause is one or more logical expressions, or predicates, that evaluate to true or false for each aggregate row retrieved from the table. Only those rows where the predicates evaluate to true are retrieved by a SELECT statement. For example, the SELECT statement below retrieves all rows where the total sales for individual companies exceed $1,000:

```
SELECT Company, SUM(sales) AS TotalSales
```

```
FROM Sales1998
GROUP BY Company
HAVING (SUM(sales) >= 1000)
ORDER BY Company
```

Multiple predicates must be separated by one of the logical operators OR or AND. Each predicate can be negated with the NOT operator. Parentheses can be used to isolate logical comparisons and groups of comparisons to produce different row evaluation criteria.

A SELECT statement can include both a WHERE clause and a HAVING clause. The WHERE clause filters the data to be aggregated, using columns not the subject of aggregate functions. The HAVING clause then further filters the data after the aggregation, using columns that are the subject of aggregate functions. The SELECT query below performs the same operation as that above, but data limited to those rows where the State column is 'CA':

```
SELECT Company, SUM(sales) AS TotalSales
FROM Sales1998
WHERE (State = 'CA')
GROUP BY Company
HAVING (TOTALSALES >= 1000)
ORDER BY Company
```

A HAVING clause filters data after the aggregation of a GROUP BY clause. For filtering based on row values prior to aggregation, use a WHERE clause.

UNION, EXCEPT, or INTERSECT Clause

The UNION clause concatenates the rows of one query result set to the end of another query result set and returns the resultant rows. The EXCEPT clause returns all of the rows from one query result set that are not present in another query result set. The INTERSECT clause returns all of the rows from one query result set that are also present in another query result set. The syntax is as follows:

```
[[UNION | EXCEPT| INTERSECT] [ALL] [SELECT...]]
```

The SELECT statement for the source and destination query result sets must include the same number of columns for them to be UNION/EXCEPT/INTERSECT-compatible. The source table structures themselve need not be the same as long as those columns included in the SELECT statements are:

```
SELECT CustNo, Company
FROM Customers
EXCEPT
SELECT OldCustNo, OldCompany
FROM Old_Customers
```

The data types for all columns retrieved by the UNION/EXCEPT/INTERSECT across the multiple query result sets must be identical. If there is a data type difference between two query result sets for a given column, an error will occur. The following query shows how to handle such a case to avoid an error:

```
SELECT S.ID, CAST(S.Date_Field AS TIMESTAMP)
FROM Source S
UNION ALL
SELECT J.ID, J.Timestamp_Field
FROM Joiner J
```

Matching names is not mandatory for result set columns retrieved by the UNION/EXCEPT/INTERSECT across the multiple query result sets. Column name differences between the multiple query result sets are automatically handled. If a column in two query result sets has a different name, the column in the UNION/EXCEPT/INTERSECTed result set will use the column name from the first SELECT statement.

By default, non-distinct rows are aggregated into single rows in a UNION/EXCEPT/INTERSECT join. Use ALL to retain non-distinct rows.

Note
When using the EXCEPT or INTERSECT clauses with the ALL keyword, the resultant rows will reflect the total counts of duplicate matching rows in both query result sets. For example, if using EXCEPT ALL with a query result set that has two 'A' rows and a query result set that has 1 'A' row, the result set will contain 1 'A' row (1 matching out of the 2). The same is true with INTERSECT. If using INTERSECT ALL with a query result set that has three 'A' rows and a query result set that has 2 'A' rows, the result set will contain 2 'A' rows (2 matching out of the 3).

To join two query result sets with UNION/EXCEPT/INTERSECT where one query does not have a column included by another, a compatible literal or expression may be used instead in the SELECT statement missing the column. For example, if there is no column in the Joining table corresponding to the Name column in Source an expression is used to provide a value for a pseudo Joining.Name column. Assuming Source.Name is of type CHAR(10), the CAST function is used to convert an empty character string to CHAR(10):

```
SELECT S.ID, S.Name
FROM Source S
INTERSECT
SELECT J.ID, CAST('' AS CHAR(10))
FROM Joiner J
```

If using an ORDER BY or TOP clause, these clauses must be specified after the last SELECT statement being joined with a UNION/EXCEPT/INTERSECT clause. The WHERE, GROUP BY, HAVING, LOCALE, ENCRYPTED, NOJOINOPTIMIZE, JOINOPTIMIZECOSTS, and NOWHEREJOINS clauses can be specified for all or some of the individual SELECT statements being joined with a UNION/EXCEPT/INTERSECT clause. The INTO clause can only be specified for the first SELECT statement in the list of unioned SELECT statements. The following example shows how you could join two SELECT statements with a UNION clause and order the final joined result set:

```
SELECT CustNo, Company
FROM Customers
UNION
SELECT OldCustNo, Company
FROM Old_Customers
ORDER BY CustNo
```

When referring to actual column names in the ORDER BY clause you must use the column name of the first SELECT statement being joined with the UNION/EXCEPT/INTERSECT clause.

ORDER BY Clause

The ORDER BY clause sorts the rows retrieved by a SELECT statement. The syntax is as follows:

```
ORDER BY column_reference [ASC|DESC]
[, column_reference...[ASC|DESC]] [NOCASE]
```

Use an ORDER BY clause to sort the rows retrieved by a SELECT statement based on the values from one or more columns. In SELECT statements, use of this clause is optional.

The value for the ORDER BY clause is a comma-separated list of column names. The columns in this list must also be in the SELECT clause of the query statement. Columns in the ORDER BY list can be from one or multiple tables. If the columns used for an ORDER BY clause come from multiple tables, the tables must all be those that are part of a join. They cannot be a table included in the statement only through a SELECT subquery.

BLOB columns cannot be used in the ORDER BY clause.

A column may be specified in an ORDER BY clause using a number representing the relative position of the column in the SELECT of the statement. Column correlation names can also be used in an ORDER BY clause columns list. Calculations cannot be used directly in an ORDER BY clause. Instead, assign a column correlation name to the calculation and use that name in the ORDER BY clause.

Use ASC (or ASCENDING) to force the sort to be in ascending order (smallest to largest), or DESC (or DESCENDING) for a descending sort order (largest to smallest). When not specified, ASC is the implied default.

Use NOCASE to force the sort to be case-insensitive. This is also useful for allowing a live result set when an index is available that matches the ORDER BY clause but is marked as case-insensitive. When not specified, case-sensitive is the implied default.

The statement below sorts the result set ascending by the year extracted from the LastInvoiceDate column, then descending by the State column, and then ascending by the uppercase conversion of the Company column:

```
SELECT EXTRACT(YEAR FROM LastInvoiceDate) AS YY,
State,
UPPER(Company)
FROM Customer
ORDER BY YY DESC, State ASC, 3
```

TOP Clause

The TOP clause cause the query to only return the top N number of rows, respecting any GROUP BY, HAVING, or ORDER BY clauses. The syntax is as follows:

```
TOP number_of_rows
```

Use a TOP clause to only extract a certain number of rows in a SELECT statement, based upon any GROUP BY, HAVING, or ORDER BY clauses. The rows that are selected start at the logical top of the result set and proceed to the total number of rows matching the TOP clause. In SELECT statements, use of the clause is optional.

LOCALE Clause

Use a LOCALE clause to set the locale of a result set created by a canned query (not live). The syntax is:

```
LOCALE locale_name | LOCALE CODE locale_code
```

If this clause is not used, the default locale of any canned result set is based upon the locale of the first table in the FROM clause of the SELECT statement. A list of locales and their IDs can be retrieved via the TDBISAMEngine GetLocaleNames method.

ENCRYPTED WITH Clause

The ENCRYPTED WITH clause causes a SELECT statement that returns a canned result set to encrypt the temporary table on disk used for the result set with the specified password. The syntax is as follows:

```
ENCRYPTED WITH password
```

Use an ENCRYPTED WITH clause to force the temporary table created by a SELECT statement that returns a canned result set to be encrypted with the specified password. This clause can also be used to encrypt the contents of a table created by a SELECT statement that uses the INTO clause.

NOJOINOPTIMIZE Clause

The NOJOINOPTIMIZE clause causes all join re-ordering to be turned off for a SELECT statement. The syntax is as follows:

```
NOJOINOPTIMIZE
```

Use a NOJOINOPTIMIZE clause to force the query optimizer to stop re-ordering joins for a SELECT statement. In certain rare cases the query optimizer might not have enough information to know that re-ordering the joins will result in worse performance than if the joins were left in their original order, so in such cases you can include this clause to force the query optimizer to not perform the join re-ordering.

JOINOPTIMIZECOSTS Clause

The JOINOPTIMIZECOSTS clause causes the optimizer to take into account I/O costs when optimizing join expressions. The syntax is as follows:

```
JOINOPTIMIZECOSTS
```

Use a JOINOPTIMIZECOSTS clause to force the query optimizer to use I/O cost projections to determine the most efficient way to process the conditions in a join expression. If you have a join expression with multiple conditions in it, then using this clause may help improve the performance of the join expression, especially if it is already executing very slowly.

NOWHEREJOINS Clause

The NOWHEREJOINS clause causes the optimizer to treat any join expressions in the WHERE clause (SQL-89-style joins) as normal, un-optimized expressions instead of inner joins. The syntax is as follows:

```
NOWHEREJOINS
```

Use a NOWHEREJOINS clause to force the query optimizer to treat any joins in the WHERE clause as normal, un-optimized expressions instead of inner joins. This is very useful when you need the conditions for filtering the results, but do not want to treat them as inner joins because they exhibit a low cardinality (there are lot of matching values). Join conditions with a low cardinality can be slow because they cause a lot of overhead in processing the sets of rows in the DBISAM engine.

4.8 INSERT Statement

Introduction

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

Syntax

```
INSERT INTO table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE]

[(columns_list)]

VALUES (update_values) | SELECT statement

[COMMIT [INTERVAL commit_interval] [FLUSH]]
```

Use the INSERT statement to add new rows of data to a single table. Use a table reference in the INTO clause to specify the table to receive the incoming data. Use the EXCLUSIVE keyword to specify that the table should be opened exclusively.

The columns list is a comma-separated list, enclosed in parentheses, of columns in the table and is optional. The VALUES clause is a comma-separated list of update values, enclosed in parentheses. Unless the source of new rows is a SELECT subquery, the VALUES clause is required and the number of update values in the VALUES clause must match the number of columns in the columns list exactly.

If no columns list is specified, incoming update values are stored in fields as they are defined sequentially in the table structure. Update values are applied to columns in the order the update values are listed in the VALUES clause. The number of update values must match the number of columns in the table exactly.

The following example inserts a single row into the Holdings table:

```
INSERT INTO Holdings
VALUES (4094095,'INPR',5000,10.500,'1998-01-02')
```

If an explicit columns list is stated, incoming update values (in the order they appear in the VALUES clause) are stored in the listed columns (in the order they appear in the columns list). NULL values are stored in any columns that are not in a columns list. When a columns list is explicitly described, there must be exactly the same number of update values in the VALUES clause as there are columns in the list.

The following example inserts a single row into the Customer table, adding data for only two of the columns in the table:

```
INSERT INTO "Customer" (CustNo, Company)
VALUES (9842,'Elevate Software, Inc.')
```

To add rows to one table that are retrieved from another table, omit the VALUES keyword and use a

subquery as the source for the new rows:

```
INSERT INTO "Customer" (CustNo, Company)
SELECT CustNo, Company
FROM "OldCustomer"
```

The INSERT statement only supports SELECT subqueries in the VALUES clause. References to tables other than the one to which rows are added or columns in such tables are only possible in SELECT subqueries.

The INSERT statement can use a single SELECT statement as the source for the new rows, but not multiple statements joined with UNION.

COMMIT Clause

The COMMIT clause is used to control how often DBISAM will commit a transaction while the INSERT statement is executing and/or whether the commit operation performs an operating system flush to disk. The INSERT statement implicitly uses a transaction if one is not already active. The default interval at which the implicit transaction is committed is based upon the record size of the table being updated in the query and the amount of buffer space available in DBISAM. The COMMIT INTERVAL clause is used to manually control the interval at which the transaction is committed based upon the number of rows inserted, and applies in both situations where a transaction was explicitly started by the application and where the transaction was implicitly started by DBISAM. In the case where a transaction was explicitly started by the application, the absence of a COMMIT INTERVAL clause in the SQL statement being executed will force DBISAM to never commit any of the effects of the SQL statement and leaves this up to the application to handle after the SQL statement completes. The syntax is as follows:

```
COMMIT [INTERVAL nnnn] [FLUSH]
```

The INTERVAL keyword is optional, allowing the application to use the default commit interval but still specify the FLUSH keyword to indicate that it wishes to have the transaction commits flushed to disk at the operating system level. Please see the Transactions and Buffering and Caching topics for more information.

Please see the Updating Tables and Query Result Sets topic for more information on adding records to a table.

4.9 UPDATE Statement

Introduction

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

Syntax

```
UPDATE table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE]

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

[FROM table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE]

[[INNER | [LEFT | RIGHT] OUTER JOIN] table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE] ON join_condition]

[WHERE predicates]

[COMMIT [INTERVAL commit_interval] [FLUSH]]

[NOJOINOPTIMIZE]
[JOINOPTIMIZECOSTS]
[NOWHEREJOINS]
```

Use the UPDATE statement to modify one or more column values in one or more existing rows in a single table per statement. Use a table reference in the UPDATE clause to specify the table to receive the data changes. Use the EXCLUSIVE keyword to specify that the table should be opened exclusively.

SET Clause

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

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

Each expression comprises the name of a column, the assignment operator (=), and the update value for that column. The update values in any one update expression may be literal values or calculated values.

FROM and JOIN Clauses

You may use an optional FROM clause with additional JOIN clauses to specify multiple tables from which an UPDATE statement retrieves data for the purpose of updating the target table. The value for a FROM clause is a comma-separated list of table names, with the first table exactly matching the table name specified after the UPDATE clause. Specified table names must follow DBISAM's SQL naming conventions

for tables. Please see the Naming Conventions topic for more information. The following UPDATE statement below updates data in one table based upon a LEFT OUTER JOIN condition to another table:

```
UPDATE orders SET ShipToContact=Customer.Contact
FROM orders LEFT OUTER JOIN customer
ON customer.custno=orders.custno
```

Note
The orders table must be specified twice - once after the UPDATE clause and again as the first table in the FROM clause.

You can use the AS keyword to specify a table correlation name, or alternately you can simply just specify the table correlation name after the source table name. The following example uses the second method to give each source table a shorter name to be used in qualifying source columns in the query:

```
UPDATE orders o SET ShipToContact=c.Contact
FROM orders o LEFT OUTER JOIN customer c
ON c.custno=o.custno
```

Use the EXCLUSIVE keyword to specify that the table should be opened exclusively.

Note
Be careful when using the EXCLUSIVE keyword with a table that is specified more than once in the same query, as is the case with recursive relationships between a table and itself.

The table reference cannot be passed to a FROM clause via a parameter. Please see the SELECT Statement topic for more information.

WHERE Clause

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

```
WHERE predicates
```

Use a WHERE clause to limit the effect of a UPDATE statement to a subset of rows in the table, and the clause is optional.

The value for a WHERE clause is one or more logical expressions, or predicates, that evaluate to TRUE or FALSE for each row in the table. Only those rows where the predicates evaluate to TRUE are modified by an UPDATE statement. For example, the UPDATE statement below modifies all rows where the State column contains a value of 'CA':

```
UPDATE SalesInfo
```

```
SET TaxRate=0.0825
WHERE (State='CA')
```

Subqueries are supported in the WHERE clause. A subquery works like a search condition to restrict the number of rows updated by the outer, or "parent" query. Such subqueries must be valid SELECT statements. SELECT subqueries cannot be correlated in DBISAM's SQL, i.e. they cannot refer to columns in the outer (or "parent") statement.

Column correlation names cannot be used in filter comparisons in the WHERE clause. Use the actual column name.

Columns devoid of data contain NULL values. To filter using such column values, use the IS NULL predicate.

The UPDATE statement may reference any table that is specified in the UPDATE, FROM, or JOIN clauses in the WHERE clause.

COMMIT Clause

The COMMIT clause is used to control how often DBISAM will commit a transaction while the UPDATE statement is executing and/or whether the commit operation performs an operating system flush to disk. The UPDATE statement implicitly uses a transaction if one is not already active. The default interval at which the implicit transaction is committed is based upon the record size of the table being updated in the query and the amount of buffer space available in DBISAM. The COMMIT INTERVAL clause is used to manually control the interval at which the transaction is committed based upon the number of rows updated, and applies in both situations where a transaction was explicitly started by the application and where the transaction was implicitly started by DBISAM. In the case where a transaction was explicitly started by the application, the absence of a COMMIT INTERVAL clause in the SQL statement being executed will force DBISAM to never commit any of the effects of the SQL statement and leaves this up to the application to handle after the SQL statement completes. The syntax is as follows:

```
COMMIT [INTERVAL nnnn] [FLUSH]
```

The INTERVAL keyword is optional, allowing the application to use the default commit interval but still specify the FLUSH keyword to indicate that it wishes to have the transaction commits flushed to disk at the operating system level. Please see the Transactions and Buffering and Caching topics for more information.

NOJOINOPTIMIZE Clause

The NOJOINOPTIMIZE clause causes all join re-ordering to be turned off for a SELECT statement. The syntax is as follows:

```
NOJOINOPTIMIZE
```

Use a NOJOINOPTIMIZE clause to force the query optimizer to stop re-ordering joins for a SELECT statement. In certain rare cases the query optimizer might not have enough information to know that re-ordering the joins will result in worse performance than if the joins were left in their original order, so in such cases you can include this clause to force the query optimizer to not perform the join re-ordering.

JOINOPTIMIZECOSTS Clause

The JOINOPTIMIZECOSTS clause causes the optimizer to take into account I/O costs when optimizing join expressions. The syntax is as follows:

```
JOINOPTIMIZECOSTS
```

Use a JOINOPTIMIZECOSTS clause to force the query optimizer to use I/O cost projections to determine the most efficient way to process the conditions in a join expression. If you have a join expression with multiple conditions in it, then using this clause may help improve the performance of the join expression, especially if it is already executing very slowly.

Please see the Updating Tables and Query Result Sets topic for more information on updating records in a table.

NOWHEREJOINS Clause

The NOWHEREJOINS clause causes the optimizer to treat any join expressions in the WHERE clause (SQL-89-style joins) as normal, un-optimized expressions instead of inner joins. The syntax is as follows:

```
NOWHEREJOINS
```

Use a NOWHEREJOINS clause to force the query optimizer to treat any joins in the WHERE clause as normal, un-optimized expressions instead of inner joins. This is very useful when you need the conditions for filtering the results, but do not want to treat them as inner joins because they exhibit a low cardinality (there are lot of matching values). Join conditions with a low cardinality can be slow because they cause a lot of overhead in processing the sets of rows in the DBISAM engine.

4.10 DELETE Statement

Introduction

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

Syntax

```
DELETE FROM table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE]

[[INNER | [LEFT | RIGHT] OUTER JOIN] table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE] ON join_condition]

[WHERE predicates]

[COMMIT [INTERVAL commit_interval] FLUSH]

[NOJOINOPTIMIZE]
[JOINOPTIMIZECOSTS]
[NOWHEREJOINS]
```

Use DELETE to delete one or more rows from one existing table per statement.

FROM Clause

The FROM clause specifies the table to use for the DELETE statement. The syntax is as follows:

```
FROM table_reference
[AS correlation_name | correlation_name] [EXCLUSIVE]
```

Specified table names must follow DBISAM's SQL naming conventions for tables. Please see the Naming Conventions topic for more information.

Use the EXCLUSIVE keyword to specify that the table should be opened exclusively.

> **Note**
> Be careful when using the EXCLUSIVE keyword with a table that is specified more than once in the same query, as is the case with recursive relationships between a table and itself.

JOIN Clauses

You may use optional JOIN clauses to specify multiple tables from which a DELETE statement retrieves data for the purpose of deleting records in the target table. The following DELETE statement below deletes data in one table based upon an INNER JOIN condition to another table:

```
DELETE FROM orders
INNER JOIN customer ON customer.custno=orders.custno
WHERE customer.country='Bermuda'
```

You can use the AS keyword to specify a table correlation name, or alternately you can simply just specify the table correlation name after the source table name. The following example uses the second method to give each source table a shorter name to be used in qualifying source columns in the query:

```
DELETE FROM orders o
INNER JOIN customer c ON c.custno=o.custno
WHERE c.country='Bermuda'
```

Please see the SELECT Statement topic for more information.

WHERE Clause

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

```
WHERE predicates
```

Use a WHERE clause to limit the effect of a DELETE statement to a subset of rows in the table, and the clause is optional.

The value for a WHERE clause is one or more logical expressions, or predicates, that evaluate to TRUE or FALSE for each row in the table. Only those rows where the predicates evaluate to TRUE are deleted by a DELETE statement. For example, the DELETE statement below deletes all rows where the State column contains a value of 'CA':

```
DELETE FROM SalesInfo
WHERE (State='CA')
```

Multiple predicates must be separated by one of the logical operators OR or AND. Each predicate can be negated with the NOT operator. Parentheses can be used to isolate logical comparisons and groups of comparisons to produce different row evaluation criteria.

Subqueries are supported in the WHERE clause. A subquery works like a search condition to restrict the number of rows deleted by the outer, or "parent" query. Such subqueries must be valid SELECT statements. SELECT subqueries cannot be correlated in DBISAM's SQL, i.e. they cannot refer to columns in the outer (or "parent") statement.

Column correlation names cannot be used in filter comparisons in the WHERE clause. Use the actual column name.

Columns devoid of data contain NULL values. To filter using such column values, use the IS NULL predicate.

The DELETE statement may reference any table that is specified in the FROM, or JOIN clauses in the WHERE clause.

COMMIT Clause

The COMMIT clause is used to control how often DBISAM will commit a transaction while the DELETE statement is executing and/or whether the commit operation performs an operating system flush to disk. The DELETE statement implicitly uses a transaction if one is not already active. The default interval at which the implicit transaction is committed is based upon the record size of the table being updated in the query and the amount of buffer space available in DBISAM. The COMMIT INTERVAL clause is used to manually control the interval at which the transaction is committed based upon the number of rows deleted, and applies in both situations where a transaction was explicitly started by the application and where the transaction was implicitly started by DBISAM. In the case where a transaction was explicitly started by the application, the absence of a COMMIT INTERVAL clause in the SQL statement being executed will force DBISAM to never commit any of the effects of the SQL statement and leaves this up to the application to handle after the SQL statement completes. The syntax is as follows:

```
COMMIT [INTERVAL nnnn] [FLUSH]
```

The INTERVAL keyword is optional, allowing the application to use the default commit interval but still specify the FLUSH keyword to indicate that it wishes to have the transaction commits flushed to disk at the operating system level. Please see the Transactions and Buffering and Caching topics for more information.

NOJOINOPTIMIZE Clause

The NOJOINOPTIMIZE clause causes all join re-ordering to be turned off for a SELECT statement. The syntax is as follows:

```
NOJOINOPTIMIZE
```

Use a NOJOINOPTIMIZE clause to force the query optimizer to stop re-ordering joins for a SELECT statement. In certain rare cases the query optimizer might not have enough information to know that re-ordering the joins will result in worse performance than if the joins were left in their original order, so in such cases you can include this clause to force the query optimizer to not perform the join re-ordering.

JOINOPTIMIZECOSTS Clause

The JOINOPTIMIZECOSTS clause causes the optimizer to take into account I/O costs when optimizing join expressions. The syntax is as follows:

```
JOINOPTIMIZECOSTS
```

Use a JOINOPTIMIZECOSTS clause to force the query optimizer to use I/O cost projections to determine the most efficient way to process the conditions in a join expression. If you have a join expression with multiple conditions in it, then using this clause may help improve the performance of the join expression, especially if it is already executing very slowly.

Please see the Updating Tables and Query Result Sets topic for more information on deleting records in a table.

NOWHEREJOINS Clause

The NOWHEREJOINS clause causes the optimizer to treat any join expressions in the WHERE clause (SQL-89-style joins) as normal, un-optimized expressions instead of inner joins. The syntax is as follows:

```
NOWHEREJOINS
```

Use a NOWHEREJOINS clause to force the query optimizer to treat any joins in the WHERE clause as normal, un-optimized expressions instead of inner joins. This is very useful when you need the conditions for filtering the results, but do not want to treat them as inner joins because they exhibit a low cardinality (there are lot of matching values). Join conditions with a low cardinality can be slow because they cause a lot of overhead in processing the sets of rows in the DBISAM engine.

4.11 CREATE TABLE Statement

Introduction

The SQL CREATE TABLE statement is used to create a table.

Syntax

```
CREATE TABLE [IF NOT EXISTS] table_reference

(

column_name data type [dimensions]
[DESCRIPTION column description]
[NULLABLE][NOT NULL]
[DEFAULT default value]
[MIN | MINIMUM minimum value]
[MAX | MAXIMUM maximum value]
[CHARCASE UPPER | LOWER | NOCHANGE]
[COMPRESS 0..9]

[, column_name...]

[, [CONSTRAINT constraint_name]
[UNIQUE] [NOCASE]
PRIMARY KEY (column_name [[ASC |ASCENDING] | [DESC | DESCENDING]]
[, column_name...])
[COMPRESS DUPBYTE | TRAILBYTE | FULL | NONE]]
[NOKEYSTATS]

[TEXT INDEX (column_name, [column_name])]
[STOP WORDS space-separated list of words]
[SPACE CHARS list of characters]
[INCLUDE CHARS list of characters]

[DESCRIPTION table_description]

[INDEX PAGE SIZE index_page_size]
[BLOB BLOCK SIZE BLOB_block_size]

[LOCALE locale_name | LOCALE CODE locale_code]

[ENCRYPTED WITH password]

[USER MAJOR VERSION user-defined_major_version]
[USER MINOR VERSION user-defined_minor_version]

[LAST AUTOINC last_autoinc_value]
)
```

Use the CREATE TABLE statement to create a table, define its columns, and define a primary key constraint.

The specified table name must follow DBISAM's SQL naming conventions for tables. Please see the

Naming Conventions topic for more information.

Column Definitions

The syntax for defining a column is as follows:

```
column_name data type [dimensions]
[DESCRIPTION column description]
[NULLABLE][NOT NULL]
[DEFAULT default value]
[MIN or MINIMUM minimum value] [MAX or MAXIMUM maximum value]
[CHARCASE UPPER | LOWER | NOCHANGE]
[COMPRESS 0..9]
```

Column definitions consist of a comma-separated list of combinations of column name, data type and (if applicable) dimensions, and optionally their description, allowance of NULL values, default value, minimum and maximum values, character-casing, and compression level (for BLOB columns). The list of column definitions must be enclosed in parentheses. The number and type of dimensions that must be specified varies with column type. Please see the Data Types and NULL Support topic for more information.

DESCRIPTION Clause

The DESCRIPTION clause specifies the description for the column. The syntax is as follows:

```
DESCRIPTION column description
```

The description must be enclosed in single or double quotes and can be any value up to 50 characters in length.

NULLABLE and NOT NULL Clauses

The NULLABLE clause specifies that the column is not required and can be NULL. The NOT NULL clause specifies that the column is required and cannot be NULL. The syntax is as follows:

```
NULLABLE
```

```
NOT NULL
```

DEFAULT Clause

The DEFAULT clause specifies the default value for the column. The syntax is as follows:

```
DEFAULT default value
```

The default value must be a value that matches the data type of the column being defined. Also, the value must be expressed in ANSI/ISO format if it is a date, time, timestamp, or number. Please see the Naming Conventions topic for more information.

MINIMUM Clause

The MINIMUM clause specifies the minimum value for the column. The syntax is as follows:

```
MIN | MINIMUM minimum value
```

The minimum value must be a value that matches the data type of the column being defined. Also, the value must be expressed in ANSI/ISO format if it is a date, time, timestamp, or number. Please see the Naming Conventions topic for more information.

MAXIMUM Clause

The MAXIMUM clause specifies the maximum value for the column. The syntax is as follows:

```
MAX | MAXIMUM maximum value
```

The maximum value must be a value that matches the data type of the column being defined. Also, the value must be expressed in ANSI/ISO format if it is a date, time, timestamp, or number. Please see the Naming Conventions topic for more information.

CHARCASE Clause

The CHARCASE clause specifies the character-casing for the column. The syntax is as follows:

```
CHARCASE UPPER | LOWER | NOCHANGE
```

If the UPPER keyword is used, then all data values in this column will be upper-cased. If the LOWER keyword is used, then all data values in this column will be lower-cased. If the NOCHANGE keyword is used, then all data values for this column will be left in their original form. This clause only applies to string columns and is ignored for all others.

The following statement creates a table with columns that include descriptions and default values:

```
CREATE TABLE employee
(
  Last_Name CHAR(20) DESCRIPTION 'Last Name',
  First_Name CHAR(15) DESCRIPTION 'First Name',
  Hire_Date DATE DESCRIPTION 'Hire Date' DEFAULT CURRENT_DATE
  Salary NUMERIC(10,2) DESCRIPTION 'Salary' DEFAULT 0.00,
  Dept_No SMALLINT DESCRIPTION 'Dept #',
  PRIMARY KEY (Last_Name, First_Name)
)
```

Primary Index Definition

Use the PRIMARY KEY (or CONSTRAINT) clause to create a primary index for the new table. The syntax is as follows:

```
[, [CONSTRAINT constraint_name]
[UNIQUE] [NOCASE]
PRIMARY KEY (column_name [[ASC |ASCENDING] | [DESC | DESCENDING]]
[, column_name...])
[COMPRESS DUPBYTE | TRAILBYTE | FULL | NONE]]
[NOKEYSTATS]
```

The columns that make up the primary index must be specified. The UNIQUE flag is completely optional and is ignored since primary indexes are always unique. The alternate CONSTRAINT syntax is also completely optional and ignored.

A primary index definition can optionally specify that the index is case-insensitive and the compression used for the index.

NOCASE Clause

The NOCASE clause specifies the that the primary index should be sorted in case-insensitive order as opposed to the default of case-sensitive order. The syntax is as follows:

```
NOCASE
```

Columns Clause

The columns clause specifies a comma-separated list of columns that make up the primary index, and optionally whether the columns should be sorted in ascending (default) or descending order. The syntax is as follows:

```
PRIMARY KEY (column_name [[ASC |ASCENDING] | [DESC | DESCENDING]]
[, column_name...])
```

The column names specified here must conform to the column naming conventions for DBISAM's SQL and must have been defined earlier in the CREATE TABLE statement. Please see the Naming Conventions topic for more information.

COMPRESS Clause

The COMPRESS clause specifies the type of index key compression to use for the primary index. The syntax is as follows:

```
COMPRESS DUPBYTE | TRAILBYTE | FULL | NONE
```

The DUPBYTE keyword specifies that duplicate-byte index key compression will be used, the TRAILBYTE keyword specifies that trailing-byte index key compression will be used, and the FULL keyword specifies that both duplicate-byte and trailing-byte index key compression will be used. The default index key compression is NONE. Please see the Index Compression topic for more information.

NOKEYSTATS Clause

The NOKEYSTATS clause specifies that the index being defined should not contain any statistics.. The syntax is as follows:

```
NOKEYSTATS
```

Under most circumstances you should not specify this clause. Not using the index statistics is only useful for very large tables where insert/update/delete performance is very important, and where it is acceptable to not have logical record numbers or statistics for optimizing filters and queries.

The following statement creates a table with a primary index on the Last_Name and First_Name columns that is case-insensitive and uses full index key compression:

```
CREATE TABLE employee
(
  Last_Name CHAR(20) DESCRIPTION 'Last Name',
  First_Name CHAR(15) DESCRIPTION 'First Name',
  Hire_Date DATE DESCRIPTION 'Hire Date' DEFAULT CURRENT_DATE
  Salary NUMERIC(10,2) DESCRIPTION 'Salary' DEFAULT 0.00,
  Dept_No SMALLINT DESCRIPTION 'Dept #',
  NOCASE PRIMARY KEY (Last_Name, First_Name) COMPRESS FULL
)
```

Note
Primary indexes are the only form of constraint that can be defined with CREATE TABLE.

Full Text Indexes Definitions

Use the TEXT INDEX, STOP WORDS, SPACE CHARS, and INCLUDE CHARS clauses (in that order) to create a full text indexes for the new table. The syntax is as follows:

```
TEXT INDEX (column_name, [column_name])
STOP WORDS space-separated list of words
SPACE CHARS list of characters
INCLUDE CHARS list of characters
```

The TEXT INDEX clause is required and consists of a comma-separated list of columns that should be full text indexed. The column names specified here must conform to the column naming conventions for DBISAM's SQL and must have been defined earlier in the CREATE TABLE statement. Please see the

Naming Conventions topic for more information.

The STOP WORDS clause is optional and consists of a space-separated list of words as a string that specify the stop words used for the full text indexes.

The SPACE CHARS and INCLUDE CHARS clauses are optional and consist of a set of characters as a string that specify the space and include characters used for the full text indexes.

For more information on how these clauses work, please see the Full Text Indexing topic.

Table Description

Use the DESCRIPTION clause to specify a description for the table. The syntax is as follows:

```
DESCRIPTION table_description
```

The description is optional and should be specified as a string.

Table Index Page Size

Use the INDEX PAGE SIZE clause to specify the index page size for the table. The syntax is as follows:

```
INDEX PAGE SIZE index_page_size
```

The index page size is optional and should be specified as an integer. Please see Appendix C - System Capacities for more information on the minimum and maximum index page sizes.

Table BLOB Block Size

Use the BLOB BLOCK SIZE clause to specify the BLOB block size for the table. The syntax is as follows:

```
BLOB BLOCK SIZE BLOB_block_size
```

The BLOB block size is optional and should be specified as an integer. Please see Appendix C - System Capacities for more information on the minimum and maximum BLOB block sizes.

Table Locale

Use the LOCALE clause to specify the locale for the table. The syntax is as follows:

```
LOCALE locale_name | LOCALE CODE locale_code
```

The locale is optional and should be specified as an identifier enclosed in double quotes ("") or square brackets ([]), if specifying a locale constant, or as an integer value, if specifying a locale ID. A list of locale constants and their IDs can be retrieved via the TDBISAMEngine GetLocaleNames method. If this clause is

not specified, then the default "ANSI Standard" locale (ID 0) will be used for the table.

Table Encryption

Use the ENCRYPTED WITH clause to specify whether the table should be encrypted with a password. The syntax is as follows:

```
ENCRYPTED WITH password
```

Table encryption is optional and the password for this clause should be specified as a string constant enclosed in single quotes ('). Please see the Encryption topic for more information.

User-Defined Versions

Use the USER MAJOR VERSION and USER MINOR VERSION clauses to specify user-defined version numbers for the table. The syntax is as follows:

```
USER MAJOR VERSION user-defined_major_version
[USER MINOR VERSION user-defined_minor_version]
```

User-defined versions are optional and the versions should be specified as integers.

Last Autoinc Value

Use the LAST AUTOINC clause to specify the last autoinc value for the table. The syntax is as follows:

```
LAST AUTOINC last_autoinc_value
```

The last autoinc value is optional and should be specified as an integer. If this clause is not specified, the default last autoinc value is 0.

Please see the Creating and Altering Tables topic for more information on creating tables.

4.12 CREATE INDEX Statement

Introduction

The SQL CREATE INDEX statement is used to create a secondary index for a table.

Syntax

```
CREATE [UNIQUE] [NOCASE]
INDEX [IF NOT EXISTS] index_name

ON table_reference

(column_name [ASC or ASCENDING | DESC or DESCENDING]
[, column_name...])
[COMPRESS DUPBYTE | TRAILBYTE | FULL | NONE]]
[NOKEYSTATS]
```

Use the CREATE INDEX statement to create a secondary index for an existing table. If index names contain embedded spaces they must be enclosed in double quotes ("") or square brackets ([]). Secondary indexes may be based on multiple columns.

UNIQUE Clause

Use the UNIQUE clause to create an index that raises an error if rows with duplicate column values are inserted. By default, indexes are not unique. The syntax is as follows:

```
UNIQUE
```

NOCASE Clause

The NOCASE clause specifies the that the secondary index should be sorted in case-insensitive order as opposed to the default of case-sensitive order. The syntax is as follows:

```
NOCASE
```

Columns Clause

The columns clause specifies a comma-separated list of columns that make up the secondary index, and optionally whether the columns should be sorted in ascending (default) or descending order. The syntax is as follows:

```
(column_name [[ASC |ASCENDING] | [DESC | DESCENDING]]
[, column_name...])
```

The column names specified here must conform to the column naming conventions for DBISAM's SQL and must have been defined earlier in the CREATE TABLE statement. Please see the Naming Conventions topic for more information.

COMPRESS Clause

The COMPRESS clause specifies the type of index key compression to use for the secondary index. The syntax is as follows:

```
COMPRESS DUPBYTE | TRAILBYTE | FULL | NONE
```

The DUPBYTE keyword specifies that duplicate-byte index key compression will be used, the TRAILBYTE keyword specifies that trailing-byte index key compression will be used, and the FULL keyword specifies that both duplicate-byte and trailing-byte index key compression will be used. The default index key compression is NONE. Please see the Index Compression topic for more information.

NOKEYSTATS Clause

The NOKEYSTATS clause specifies that the index being defined should not contain any statistics.. The syntax is as follows:

```
NOKEYSTATS
```

Under most circumstances you should not specify this clause. Not using the index statistics is only useful for very large tables where insert/update/delete performance is very important, and where it is acceptable to not have logical record numbers or statistics for optimizing filters and queries.

The following statement creates a multi-column secondary index that sorts in ascending order for the CustNo column and descending order for the SaleDate column:

```
CREATE INDEX CustDate
ON Orders (CustNo, SaleDate DESC) COMPRESS DUPBYTE
```

The following statement creates a unique, case-insensitive secondary index:

```
CREATE UNIQUE NOCASE INDEX "Last Name"
ON Employee (Last_Name) COMPRESS FULL
```

Please see the Adding and Deleting Indexes from a Table topic for more information on creating indexes.

4.13 ALTER TABLE Statement

Introduction

The SQL ALTER TABLE statement is used to restructure a table.

Syntax

```
ALTER TABLE [IF EXISTS] table_reference

[[ADD [COLUMN] [IF NOT EXISTS]
column_name data type [dimensions]
[AT column_position]
[DESCRIPTION column description]
[NULLABLE][NOT NULL]
[DEFAULT default value]
[MIN or MINIMUM minimum value]
[MAX or MAXIMUM maximum value]
[CHARCASE UPPER | LOWER | NOCHANGE]
[COMPRESS 0..9]]

|

[REDEFINE [COLUMN] [IF EXISTS]
column_name [new_column_name] data type [dimensions]
[AT column_position]
[DESCRIPTION column description]
[NULLABLE][NOT NULL]
[DEFAULT default value]
[MIN or MINIMUM minimum value]
[MAX or MAXIMUM maximum value]
[CHARCASE UPPER | LOWER | NOCHANGE]
[COMPRESS 0..9]]

|

[DROP [COLUMN] [IF EXISTS] column_name]]

[, ADD [COLUMN] column_name
REDEFINE [COLUMN] column_name
DROP [COLUMN] column_name...]

[, ADD [CONSTRAINT constraint_name]
[UNIQUE] [NOCASE] PRIMARY KEY
(column_name [ASC or ASCENDING | DESC or DESCENDING]
[, column_name...])
[COMPRESS DUPBYTE | TRAILBYTE | FULL | NONE]]
[NOKEYSTATS]

[, REDEFINE [CONSTRAINT constraint_name]
[UNIQUE] [NOCASE] PRIMARY KEY
(column_name [ASC or ASCENDING | DESC or DESCENDING]
[, column_name...])
[COMPRESS DUPBYTE | TRAILBYTE | FULL | NONE]]
[NOKEYSTATS]
```

```
[, DROP [CONSTRAINT constraint_name] PRIMARY KEY]

[TEXT INDEX (column_name, [column_name])]
[STOP WORDS space-separated list of words]
[SPACE CHARS list of characters]
[INCLUDE CHARS list of characters]

[DESCRIPTION table_description]

[INDEX PAGE SIZE index_page_size]
[BLOB BLOCK SIZE BLOB_block_size]

[LOCALE locale_name | LOCALE CODE locale_code]

[ENCRYPTED WITH password]

[USER MAJOR VERSION user-defined_major_version]
[USER MINOR VERSION user-defined_minor_version]

[LAST AUTOINC last_autoinc_value]

[NOBACKUP]
```

Use the ALTER TABLE statement to alter the structure of an existing table. It is possible to delete one column and add another in the same ALTER TABLE statement as well as redefine an existing column without having to first drop the column and then re-add the same column name. This is what is sometimes required with other database engines and can result in loss of data. DBISAM's REDEFINE keyword removes this problem. In addition, the IF EXISTS and IF NOT EXISTS clauses can be used with the ADD, REDEFINE, and DROP keywords to allow for action on columns only if they do or do not exist.

The DROP keyword requires only the name of the column to be deleted. The ADD keyword requires the same combination of column name, data type and possibly dimensions, and extended column definition information as the CREATE TABLE statement when defining new columns.

The statement below deletes the column FullName and adds the column LastName, but only if the LastName column doesn't already exist:

```
ALTER TABLE Names
DROP FullName,
ADD IF NOT EXISTS LastName CHAR(25)
```

It is possible to delete and add a column of the same name in the same ALTER TABLE statement, however any data in the column is lost in the process. An easier way is to use the extended syntax provided by DBISAM's SQL with the REDEFINE keyword:

```
ALTER TABLE Names
REDEFINE LastName CHAR(30)
```

Note
In order to remove the full text index completely, you would specify no columns in the TEXT INDEX clause like this:

```
ALTER TABLE Customer
TEXT INDEX ()
```

NOBACKUP Clause

The NOBACKUP clause specifies that no backup files should be created during the process of altering the table's structure.

Please see the CREATE TABLE statement for more information on all other clauses used in the ALTER TABLE statement. Their usage is the same as with the CREATE TABLE statement.

Please see the Creating and Altering Tables topic for more information on altering the structure of tables.

4.14 EMPTY TABLE Statement

Introduction

The SQL EMPTY TABLE statement is used to empty a table of all data.

Syntax

```
EMPTY TABLE [IF EXISTS] table_reference
```

Use the EMPTY TABLE statement to remove all data from an existing table. The statement below empties a table:

```
EMPTY TABLE Employee
```

Please see the Emptying Tables topic for more information on emptying tables.

4.15 OPTIMIZE TABLE Statement

Introduction

The SQL OPTIMIZE TABLE statement is used to optimize a table.

Syntax

```
OPTIMIZE TABLE [IF EXISTS] table_reference

[ON index_name]

[NOBACKUP]
```

Use the OPTIMIZE TABLE statement to remove all free space from a table and organize the data more efficiently.

ON Clause

The ON clause is optional and specifies the name of an index in the table to use for organizing the physical data records. It is usually recommended that you do not specify this clause, which will result in the table being organized using the primary index.

NOBACKUP Clause

The NOBACKUP clause specifies that no backup files should be created during the process of optimizing the table.

The statement below optimizes a table and suppresses any backup files:

```
OPTIMIZE TABLE Employee NOBACKUP
```

Please see the Optimizing Tables topic for more information on optimizing tables.

4.16 EXPORT TABLE Statement

Introduction

The SQL EXPORT TABLE statement is used to export a table to a delimited text file.

Syntax

```
EXPORT TABLE [IF EXISTS] table_reference [EXCLUSIVE]

TO text_file_name

[DELIMITER delimiter_character]

[WITH HEADERS]

[COLUMNS (column_name [, column_name])]

[DATE date_format]
[TIME time_format]
[DECIMAL decimal_separator]
```

Use the EXPORT TABLE statement to export a table to a delimited text file specified by the TO clause. The file name must be enclosed in double quotes ("") or square brackets ([]) if it contains a drive, path, or file extension. Use the EXCLUSIVE keyword to specify that the table should be opened exclusively.

DELIMITER Clause

The DELIMITER clause is optional and specifies the delimiter character to use in the exported text file. The DELIMITER character should be specified as a single character constant enclosed in single quotes (') or specified using the pound (#) sign and the ASCII character value. The default delimiter character is the comma (,).

WITH HEADERS Clause

The WITH HEADERS clause is optional and specifies that the exported text file should contain column headers for all columns as the first row.

COLUMNS Clause

The columns clause is optional and specifies a comma-separated list of columns that should be exported to the text file. The column names specified here must conform to the column naming conventions for DBISAM's SQL and must exist in the table being exported. Please see the Naming Conventions topic for more information.

DATE, TIME, and DECIMAL Clauses

The DATE, TIME, and DECIMAL clauses are optional and specify the formats and decimal separator that should be used when exporting dates, times, timestamps, and numbers. The DATE and TIME formats

should be specified as string constants enclosed in single quotes ('') and the DECIMAL separator should be specified as a single character constant enclosed in single quotes ('') or specified using the pound (#) sign and the ASCII character value. The default date format is 'yyyy-mm-dd', the default time format is 'hh:mm:ss.zzz ampm', and the default decimal separator is '.'.

The statement below exports three fields from the Employee table into a file called 'employee.txt':

```
EXPORT TABLE Employee
TO "c:\mydata\employee.txt"
WITH HEADERS
COLUMNS (ID, FirstName, LastName)
```

Please see the Importing and Exporting Tables and Query Result Sets topic for more information on exporting tables.

4.17 IMPORT TABLE Statement

Introduction

The SQL IMPORT TABLE statement is used to import data from delimited text file into a table.

Syntax

```
IMPORT TABLE [IF EXISTS] table_reference

FROM text_file_name

[DELIMITER delimiter_character]

[WITH HEADERS]

[COLUMNS (column_name [, column_name])]

[DATE date_format]
[TIME time_format]
[DECIMAL decimal_separator]
```

Use the IMPORT TABLE statement to import data into a table from a delimited text file specified by the FROM clause. The file name must be enclosed in double quotes ("") or square brackets ([]) if it contains a drive, path, or file extension. Use the EXCLUSIVE keyword to specify that the table should be opened exclusively.

DELIMITER Clause

The DELIMITER clause is optional and specifies the delimiter character used in the imported text file to separate data from different columns. The DELIMITER character should be specified as a single character constant enclosed in single quotes (') or specified using the pound (#) sign and the ASCII character value. The default delimiter character is the comma (,).

WITH HEADERS Clause

The WITH HEADERS clause is optional and specifies that the imported text file contains column headers for all columns as the first row. In such a case DBISAM will not import this row as a record but will instead ignore it.

COLUMNS Clause

The columns clause is optional and specifies a comma-separated list of columns that the imported text file contains. If the imported text file contains column data in a different order than that of the table, or only a subset of column data, then it is very important that this clause be used. Also, the column names specified here must conform to the column naming conventions for DBISAM's SQL and must exist in the table being exported. Please see the Naming Conventions topic for more information.

DATE, TIME, and DECIMAL Clauses

The DATE, TIME, and DECIMAL clauses are optional and specify the formats and decimal separator that should be used when importing dates, times, timestamps, and numbers from the text file. The DATE and TIME formats should be specified as string constants enclosed in single quotes (') and the DECIMAL separator should be specified as a single character constant enclosed in single quotes (') or specified using the pound (#) sign and the ASCII character value. The default date format is 'yyyy-mm-dd', the default time format is 'hh:mm:ss.zzz ampm', and the default decimal separator is '.'.

The statement below imports three fields from a file called 'employee.txt' into the Employee table:

```
IMPORT TABLE Employee
FROM "c:\mydata\employee.txt"
WITH HEADERS
COLUMNS (ID, FirstName, LastName)
```

Please see the Importing and Exporting Tables and Query Result Sets topic for more information on importing tables.

4.18 VERIFY TABLE Statement

Introduction

The SQL VERIFY TABLE statement is used to verify a table and make sure that there is no corruption in the table.

Syntax

```
VERIFY TABLE [IF EXISTS] table_reference
```

Use the VERIFY TABLE statement to verify the physical structure of a table to make sure that it is not corrupted. The statement below verifies a table:

```
VERIFY TABLE Employee
```

You can use the REPAIR TABLE SQL statement to repair a table that is determined to be corrupted via the VERIFY TABLE statement.

Please see the Verifying and Repairing Tables topic for more information on verifying tables.

4.19 REPAIR TABLE Statement

Introduction

The SQL REPAIR TABLE statement is used to repair a table that is corrupted or suspected of being corrupted.

Syntax

```
REPAIR TABLE [IF EXISTS] table_reference

FORCEINDEXREBUILD
```

Use the REPAIR TABLE statement to repair the physical structure of a table that is corrupted or suspected of being corrupted.

FORCEINDEXREBUILD Clause

Use the FORCEINDEXREBUILD clause to force the indexes in the table to be rebuilt regardless of whether they are determined to be corrupted or not. Sometimes there is corruption in indexes that DBISAM cannot detect in the table verification or repair process, and this clause will resolve such an issue.

The statement below repairs a table:

```
REPAIR TABLE Employee
```

You can use the VERIFY TABLE SQL statement to verify a table and determine if it is corrupted.

Please see the Verifying and Repairing Tables topic for more information on repairing tables.

4.20 UPGRADE TABLE Statement

Introduction

The SQL UPGRADE TABLE statement is used to upgrade a table from a previous DBISAM table format to the current table format.

Syntax

```
UPGRADE TABLE [IF EXISTS] table_reference
```

Use the UPGRADE TABLE statement to upgrade a table to the current DBISAM table format. The statement below upgrades a table:

```
UPGRADE TABLE Employee
```

Please see the Upgrading Tables topic for more information on upgrading tables.

4.21 DROP TABLE Statement

Introduction

The SQL DROP TABLE statement is used to delete a table.

Syntax

```
DROP TABLE [IF EXISTS] table_reference
```

Use the DROP TABLE statement to delete an existing table. The statement below drops a table:

```
DROP TABLE Employee
```

Please see the Deleting Tables topic for more information on deleting tables.

4.22 RENAME TABLE Statement

Introduction

The SQL RENAME TABLE statement is used to rename a table.

Syntax

```
RENAME TABLE [IF EXISTS] table_reference
TO table_reference
```

Use the RENAME TABLE statement to rename a table. The statement below renames a table:

```
RENAME TABLE Employee
TO Employees
```

Please see the Renaming Tables topic for more information on renaming tables.

4.23 DROP INDEX Statement

Introduction

The SQL DROP INDEX statement is used to delete a primary or secondary index from a table.

Syntax

```
DROP INDEX [IF EXISTS]
table_reference.index_name | PRIMARY
```

Use the DROP INDEX statement to delete a primary or secondary index. To delete a secondary index, identify the index using the table name and index name separated by an identifier connector symbol (.):

```
DROP INDEX Employee."Last Name"
```

To delete a primary index, identify the index with the keyword PRIMARY:

```
DROP INDEX Orders.PRIMARY
```

Please see the Adding and Deleting Indexes from a Table topic for more information on deleting indexes.

4.24 START TRANSACTION Statement

Introduction

The SQL START TRANSACTION statement is used to start a transaction on the current database.

Syntax

```
START TRANSACTION
[WITH <comma-separated list of tables>]
```

Use the START TRANSACTION statement to start a transaction. The WITH clause allows to to start a restricted transaction on a specified set of tables. It accepts a comma-delimited list of table names to include in the restricted transaction.

Please see the Transactions topic for more information on transactions.

4.25 COMMIT Statement

Introduction

The SQL COMMIT statement is used to commit an active transaction on the current database.

Syntax

```
COMMIT [WORK] [FLUSH]
```

Use the COMMIT statement to commit an active transaction. You may optionally include the WORK keyword for compatibility with the SQL standard.

FLUSH Clause

Use the FLUSH clause to indicate that the commit operation should also instruct the operating system to flush all committed data to disk.

Please see the Transactions topic for more information on transactions.

4.26 ROLLBACK Statement

Introduction

The SQL ROLLBACK statement is used to rollback an active transaction on the current database.

Syntax

```
ROLLBACK [WORK]
```

Use the ROLLBACK statement to rollback an active transaction. You may optionally include the WORK keyword for compatibility with the SQL standard.

Please see the Transactions topic for more information on transactions.

This page intentionally left blank

Chapter 5
Component Reference

5.1 EDBISAMEngineError Component

Header File: dbisamtb

Inherits From Db

An EDBISAMEngineError exception object is raised whenever a DBISAM error occurs. You will find a list of all DBISAM error codes along with instructions on how to change the default error messages in Appendix B - Error Codes and Messages in this manual. For just general information on exception handling in DBISAM please see the Exception Handling and Errors topic in this manual.

Properties	Methods	Events
ErrorCode	EDBISAMEngineError	
ErrorColumn		
ErrorDatabaseName		
ErrorEventName		
ErrorFieldName		
ErrorIndexName		
ErrorLine		
ErrorMessage		
ErrorProcedureName		
ErrorRemoteName		
ErrorTableName		
ErrorUserName		
OSErrorCode		
SocketErrorCode		

EDBISAMEngineError.ErrorCode Property

```
__property System::Word ErrorCode
```

Indicates the native DBISAM error code being raised in the current exception.

Note
This property is always set for every exception.

EDBISAMEngineError.ErrorColumn Property

```
__property int ErrorColumn
```

Indicates the column of text in that the current exception applies to.

Note
This property may or may not be set depending upon the exception being raised.

EDBISAMEngineError.ErrorDatabaseName Property

```
__property System::AnsiString ErrorDatabaseName
```

Indicates the database name that the current exception applies to.

Note
This property may or may not be set depending upon the exception being raised.

EDBISAMEngineError.ErrorEventName Property

```
__property System::AnsiString ErrorEventName
```

Indicates the scheduled event name that the current exception applies to.

Note
This property may or may not be set depending upon the exception being raised.

EDBISAMEngineError.ErrorFieldName Property

```
__property System::AnsiString ErrorFieldName
```

Indicates the field or column name that the current exception applies to.

Note
This property may or may not be set depending upon the exception being raised.

EDBISAMEngineError.ErrorIndexName Property

```
__property System::AnsiString ErrorIndexName
```

Indicates the index name that the current exception applies to. This property will be set to 'Primary' if the exception refers to the primary index of a table.

Note
This property may or may not be set depending upon the exception being raised.

EDBISAMEngineError.ErrorLine Property

```
__property int ErrorLine
```

Indicates the line of text in that the current exception applies to.

Note
This property may or may not be set depending upon the exception being raised.

EDBISAMEngineError.ErrorMessage Property

```
__property System::AnsiString ErrorMessage
```

Indicates the extended error message that gives further information on the exception.

Note
This property may or may not be set depending upon the exception being raised.

EDBISAMEngineError.ErrorProcedureName Property

```
__property System::AnsiString ErrorProcedureName
```

Indicates the server-side procedure name that the current exception applies to.

Note
This property may or may not be set depending upon the exception being raised.

EDBISAMEngineError.ErrorRemoteName Property

```
__property System::AnsiString ErrorRemoteName
```

Indicates the database server host name or IP address that the current exception applies to.

Note
This property may or may not be set depending upon the exception being raised.

EDBISAMEngineError.ErrorTableName Property

```
__property System::AnsiString ErrorTableName
```

Indicates the table name that the current exception applies to.

Note
This property may or may not be set depending upon the exception being raised.

EDBISAMEngineError.ErrorUserName Property

```
__property System::AnsiString ErrorUserName
```

Indicates the user name that the current exception applies to.

Note
This property may or may not be set depending upon the exception being raised.

EDBISAMEngineError.OSErrorCode Property

```
__property int OSErrorCode
```

Indicates the last operating system-specific error code logged by the operating system. This property is always set, although it may not always contain a non-0 value. This property can be useful for debugging unusual error conditions coming from the operating system.

EDBISAMEngineError.SocketErrorCode Property

```
__property int SocketErrorCode
```

Indicates the last operating system-specific TCP/IP socket error code logged by the operating system. This property is always set, although it may not always contain a non-0 value. This property can be useful for debugging unusual error conditions coming from the operating system's TCP/IP socket subsystem.

EDBISAMEngineError.EDBISAMEngineError Method

```
__fastcall EDBISAMEngineError(System::Word ErrorCode)

inline __fastcall EDBISAMEngineError(const System::UnicodeString
      Msg, System::TVarRec const *Args, const int Args_Size) :
      Data::Db::EDatabaseError(Msg, Args, Args_Size) { }

inline __fastcall EDBISAMEngineError(NativeUInt Ident) :
      Data::Db::EDatabaseError(Ident) { }

inline __fastcall EDBISAMEngineError(System::PResStringRec
      ResStringRec) : Data::Db::EDatabaseError(ResStringRec) { }

inline __fastcall EDBISAMEngineError(NativeUInt Ident,
      System::TVarRec const *Args, const int Args_Size) :
      Data::Db::EDatabaseError(Ident, Args, Args_Size) { }

inline __fastcall EDBISAMEngineError(System::PResStringRec
      ResStringRec, System::TVarRec const *Args, const int Args_Size)
      : Data::Db::EDatabaseError(ResStringRec, Args, Args_Size) { }

inline __fastcall EDBISAMEngineError(const System::UnicodeString
      Msg, int AHelpContext) : Data::Db::EDatabaseError(Msg,
      AHelpContext) { }

inline __fastcall EDBISAMEngineError(const System::UnicodeString
      Msg, System::TVarRec const *Args, const int Args_Size, int
      AHelpContext) : Data::Db::EDatabaseError(Msg, Args, Args_Size,
      AHelpContext) { }

inline __fastcall EDBISAMEngineError(NativeUInt Ident, int
      AHelpContext) : Data::Db::EDatabaseError(Ident, AHelpContext) {
      }

inline __fastcall EDBISAMEngineError(System::PResStringRec
      ResStringRec, int AHelpContext) :
      Data::Db::EDatabaseError(ResStringRec, AHelpContext) { }

inline __fastcall EDBISAMEngineError(System::PResStringRec
      ResStringRec, System::TVarRec const *Args, const int Args_Size,
      int AHelpContext) : Data::Db::EDatabaseError(ResStringRec, Args,
      Args_Size, AHelpContext) { }

inline __fastcall EDBISAMEngineError(NativeUInt Ident,
      System::TVarRec const *Args, const int Args_Size, int
      AHelpContext) : Data::Db::EDatabaseError(Ident, Args, Args_Size,
      AHelpContext) { }
```

Creates an instance of EDBISAMEngineError using a specified DBISAM engine error code. The constructor calls the constructor method inherited from Exception (using an empty string) to construct an initialized instance of EDBISAMEngineError.

5.2 TDBISAMBaseDataSet Component

Header File: dbisamtb

Inherits From Db

The TDBISAMBaseDataSet component is a dataset component that provides a base component for any DBISAM dataset. Applications never use TDBISAMBaseDataSet components directly. Instead they use the descendants of TDBISAMBaseDataSet, the TDBISAMQuery and TDBISAMTable components, which inherit its database-related properties and methods.

Properties	Methods	Events
	TDBISAMBaseDataSet	

TDBISAMBaseDataSet.TDBISAMBaseDataSet Method

```
inline __fastcall virtual
      TDBISAMBaseDataSet(System::Classes::TComponent* AOwner) :
      Data::Db::TDataSet(AOwner) { }
```

Use the New operator to create an instance of a TDBISAMBaseDataSet component using the constructor. However, you should not use this constructor to create an instance of this component directly. Instead you should create an instance of a TDBISAMTable or TDBISAMQuery component, which both descend from this component and provide a published interface for use at design-time.

5.3 TDBISAMBlobStream Component

Header File: dbisamtb

Inherits From Classes

Use the TDBISAMBlobStream object to access or modify the contents of a BLOB field in a dataset using a stream interface. BLOB fields include TBlobField objects and descendants of TBlobField such as TGraphicField and TMemoField objects. TBlobField objects use streams to implement many of their data access properties and methods via the standard CreateBlobStream method that is implemented by the DBISAM dataset components.

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

Note
For proper results when updating a BLOB field using a TDBISAMBlobStream object, you must create the TDBISAMBlobStream object after calling the Append/Insert or Edit method for the dataset containing the BLOB field. Also, you must free the TDBISAMBlobStream object before calling the Post method to post the changes to the dataset. Finally, be sure to use the proper open mode when creating a TDBISAMBlobStream object for updating (either bmReadWrite or bmWrite).

Properties	Methods	Events
	Read	
	Seek	
	TDBISAMBlobStream	
	Truncate	
	Write	

TDBISAMBlobStream.Read Method

```
virtual int __fastcall Read(void *Buffer, int Count)
```

Read transfers up to Count bytes from the BLOB field into Buffer, starting in the current position, and then advances the current position by the number of bytes actually transferred. Read returns the number of bytes actually transferred (which may be less than the number requested in Count.) Buffer must have at least Count bytes allocated to hold the data that was read from the field.

Read ignores the Transliterate property of the field since DBISAM always reads data using the ANSI character set.

All the other data-reading methods of a TDBISAMBlobStream object (ReadBuffer, ReadComponent) call Read to do their actual reading.

Note
Do not call Read when the TDBISAMBlobStream object was created in bmWrite mode.

TDBISAMBlobStream.Seek Method

```
virtual int __fastcall Seek(int Offset, System::Word Origin)
```

Use Seek to move the current position within the BLOB field by the indicated offset. Seek allows an application to read from or write to a particular location within the BLOB field.

The Origin parameter indicates how to interpret the Offset parameter. Origin should be one of the following values:

Origin	Description
soFromBeginning	Offset is from the beginning of the BLOB field. Seek moves to the position Offset. Offset must be >= 0.
soFromCurrent	Offset is from the current position in the BLOB field. Seek moves to Position + Offset.
soFromEnd	Offset is from the end of the BLOB field. Offset must be <= 0 to indicate a number of bytes before the end of the BLOB.

Seek returns the new value of the Position property, the new current position in the BLOB field.

TDBISAMBlobStream.TDBISAMBlobStream Method

```
__fastcall TDBISAMBlobStream(Data::Db::TBlobField* Field,
      Data::Db::TBlobStreamMode Mode)
```

Call the constructor to obtain an instance of TDBISAMBlobStream for reading from or writing to a specific TBlobField object. The constructor links the TDBISAMBlobStream to the field object specified by the Field parameter. Mode specifies whether the stream will be used to read data (bmRead), write data (bmWrite) or modify data (bmReadWrite).

TDBISAMBlobStream.Truncate Method

```
void __fastcall Truncate(void)
```

Use Truncate to limit the size of the BLOB field. Calling Truncate when the current position is 0 will clear the contents of the BLOB field.

Note
Do not call Truncate when the TDBISAMBlobStream was created in bmRead mode.

TDBISAMBlobStream.Write Method

```
virtual int __fastcall Write(const void *Buffer, int Count)
```

Use Write to write Count bytes to the BLOB field, starting at the current position. The Write method for TDBISAMBlobStream always writes the entire Count bytes, as a BLOB field does not necessarily include a termination character. Thus, Write is equivalent to the WriteBuffer method.

Write ignores the Transliterate property of the field since DBISAM always writes data using the ANSI character set.

All the other data-writing methods of a TDBISAMBlobStream object (WriteBuffer, WriteComponent) call Write to do their actual writing.

Note
Do not call Write when the TDBISAMBlobStream object was created in bmRead mode.

5.4 TDBISAMDatabase Component

Header File: dbisamtb

Inherits From Db

Use the TDBISAMDatabase component to manage a local or remote database within an application. A database serves as a container for tables and allows for transaction control for multiple table updates as well as easy online backup and restore of all tables contained within it.

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

Also, you may have multiple TDBISAMDatabase components referring to the same local or remote database and they will share the same transaction status, etc.

Properties	Methods	Events
DatabaseName	Backup	OnBackupLog
DataSets	BackupInfo	OnBackupProgress
Directory	CloseDataSets	OnCommit
EngineVersion	Commit	OnRestoreLog
Handle	Execute	OnRestoreProgress
InTransaction	Restore	OnRollback
KeepConnection	Rollback	OnStartTransaction
KeepTablesOpen	StartTransaction	
RemoteDatabase	TDBISAMDatabase	
Session	ValidateName	
SessionName		
StoreConnected		
Temporary		

TDBISAMDatabase.DatabaseName Property

```
__property System::UnicodeString DatabaseName
```

Use to the DatabaseName property to specify the name of the database to associate with this TDBISAMDatabase component. The database name is arbitrary and is used only for identification of the database when connecting TDBISAMTable and TDBISAMQuery components. It is best to think of the DatabaseName as an alias to the physical location of the database tables, which is represented by the Directory property for local sessions and the RemoteDatabase property for remote sessions. The DatabaseName property must begin with an alpha character.

Note
Attempting to set this property when the Connected property of the TDBISAMDatabase component is True will result in an exception being raised.

TDBISAMDatabase.DataSets Property

```
__property TDBISAMDBDataSet* DataSets[int Index]
```

The DataSets property provides an indexed array of all active datasets for a TDBISAMDatabase component. An active dataset is one that is currently open.

Note
A "dataset" is either a TDBISAMTable or TDBISAMQuery component, both of which descend from the TDBISAMDBDataSet component.

TDBISAMDatabase.Directory Property

```
__property System::UnicodeString Directory
```

Use the Directory property to specify the operating system directory where the database tables are located for a TDBISAMDatabase component. This property only applies to TDBISAMDatabase components that are connected to a local TDBISAMSession component whose SessionType property is set to stLocal. With local sessions a database is synonymous with a directory. For TDBISAMDatabase components that are connected to remote sessions you should use the RemoteDatabase property to specify the database.

It is not recommended that you leave this property blank since this will cause the TDBISAMDatabase component to look in the current working directory for database tables, and the current working directory may change frequently during the execution of an application.

Note
Attempting to set this property when the Connected property of the TDBISAMDatabase component is True will result in an exception being raised.

TDBISAMDatabase.EngineVersion Property

```
__property System::UnicodeString EngineVersion
```

Indicates the current version of DBISAM being used. This property is read-only, but published so that it is visible in the Object Inspector in Delphi, Kylix, and C++Builder.

TDBISAMDatabase.Handle Property

```
__property Dbisamen::TDBISAMDatabaseManager* Handle
```

The Handle property is for internal use only and is not useful to the application developer using DBISAM.

TDBISAMDatabase.InTransaction Property

```
__property bool InTransaction
```

Use the InTransaction property at run-time to determine if a transaction is currently in progress. The InTransaction property is True if a transaction is in progress and False if a transaction is not in progress.

The value of the InTransaction property cannot be changed directly. Calling the TDBISAMDatabase StartTransaction sets the InTransaction property to True. Calling the TDBISAMDatabase Commit or Rollback methods sets the InTransaction property to False.

Note
If the current TDBISAMDatabase component is sharing its internal handle with another TDBISAMDatabase component, then calling StartTransaction on one component will also cause the other component's InTransaction property to reflect True. The same holds true for two TDBISAMDatabase components that refer to the same local or remote database since DBISAM never allocates more than one internal handle for a given database.

TDBISAMDatabase.KeepConnection Property

```
__property bool KeepConnection
```

Use the KeepConnection property to specify whether an application remains connected to a database even if no datasets are open. When the KeepConnection property is True (the default) the connection is maintained. When the KeepConnection property 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.

Note
The KeepConnection property setting for temporary TDBISAMDatabase components created automatically as needed is determined by the KeepConnections property of the TDBISAMSession component that the TDBISAMDatabase component is linked to.

TDBISAMDatabase.KeepTablesOpen Property

```
__property bool KeepTablesOpen
```

Use the KeepTablesOpen property to specify that any tables opened for shared (non-exclusive) use are kept open internally in DBISAM, even though they have been closed by the application. These tables are kept open internally until the TDBISAMDatabase component is disconnected and the Active property is False. This can result in significant performance improvements in situations where DBISAM must open and close the same set of tables frequently, such as with large SQL scripts.

However, use this property very carefully since it can cause access problems that are hard to diagnose. For example, you may try to alter the structure of a table that is internally still open in DBISAM and the resulting DBISAM_OSEACCES error message issued by DBISAM could be very confusing. In situations like this, disconnecting the TDBISAMDatabase component will solve the problem.

Note
Attempting to set this property when the Connected property of the TDBISAMDatabase component is True will result in an exception being raised.

TDBISAMDatabase.RemoteDatabase Property

```
__property System::UnicodeString RemoteDatabase
```

Use the RemoteDatabase property to specify the database name on the database server where the database tables are located for a TDBISAMDatabase component. This property only applies to TDBISAMDatabase components that are connected to a remote TDBISAMSession component whose SessionType property is set to stRemote. With remote sessions a database is synonymous with a logical database name on a database server. For TDBISAMDatabase components that are connected to local sessions you should use the Directory property to specify the database.

Note
Attempting to set this property when the Connected property of the TDBISAMDatabase component is True will result in an exception being raised.

TDBISAMDatabase.Session Property

```
__property TDBISAMSession* Session
```

Use the Session property to determine the TDBISAMSession component that the TDBISAMDatabase component is linked to. By default, a TDBISAMDatabase component is linked with the default TDBISAMSession component that is automatically created for all applications (it can be referenced via the global Session function in the dbisamtb unit (Delphi and Kylix) or the dbisamtb header file (C++Builder). To assign a TDBISAMDatabase component to a different session in a multi-threaded application, specify the name of a different TDBISAMSession component in the SessionName property.

TDBISAMDatabase.SessionName Property

```
__property System::UnicodeString SessionName
```

Use the SessionName property to specify the session with which the TDBISAMDatabase component is linked. If the SessionName property is blank, a TDBISAMDatabase component is automatically linked with the default TDBISAMSession component that can be referenced via the global Session function in the dbisamtb unit (Delphi and Kylix) or the dbisamtb header file (C++Builder). To link a TDBISAMDatabase component with a different session in an application, the SessionName property must match the SessionName property of an existing TDBISAMSession component.

TDBISAMDatabase.StoreConnected Property

```
__property bool StoreConnected
```

Use the StoreConnected property to determine if the database should store the current value of its Connected property, and subsequently, the Active property values of all other DBISAM components such as the TDBISAMTable, and TDBISAMQuery components, in the owner form or data module. The default value for this property is True.

TDBISAMDatabase.Temporary Property

```
__property bool Temporary
```

The Temporary property indicates whether a TDBISAMDatabase component is temporary and created by DBISAM as needed, or persistent and explicitly created, managed, and freed within the application. A temporary TDBISAMDatabase component is created when a dataset is opened and the dataset is not already linked with an existing TDBISAMDatabase component via its DatabaseName property. If Temporary remains True, then a temporary TDBISAMDatabase component is freed when the dataset is closed. An application can prevent the destruction of a temporary TDBISAMDatabase component by setting Temporary to False while the dataset is active, but the application is then responsible for closing the TDBISAMDatabase component when it is no longer needed.

Note
A "dataset" is either a TDBISAMTable or TDBISAMQuery component, both of which descend from the TDBISAMDBDataSet component.

TDBISAMDatabase.Backup Method

```
bool __fastcall Backup(const System::UnicodeString BackupName,
      const System::UnicodeString BackupDescription, System::Byte
      Compression, System::Classes::TStrings* BackupTables)
```

Call the Backup method to backup all tables specified by the BackupTables parameter into the backup file specified by the BackupName parameter. You can specify a description for the backup with the BackupDescription parameter. The Compression parameter is specified as a Byte value between 0 and 9, with the default being 0, or none, and 6 being the best selection for size/speed. The default compression is ZLib, but can be replaced by using the TDBISAMEngine events for specifying a different type of compression. Please see the Compression and Customizing the Engine topics for more information.

The Backup method cannot be run when a transaction is currently active for the database. You can inspect the InTransaction property to determine if a transaction is currently active for the database. When the backup executes, it obtains a read lock for the entire database that prevents any sessions from performing any writes to any of the tables in the database until the backup completes.

Note
The BackupName parameter can contain a full path and file name, however when calling this method from within a remote session for a remote database you must make sure that the path is relative to the database server, not the client workstation. The best solution is to run the backup from a scheduled event on the server. Please see the Customizing the Engine topic for more information.

TDBISAMDatabase.BackupInfo Method

```
bool __fastcall BackupInfo(const System::UnicodeString
      BackupName, System::UnicodeString &BackupDescription,
      System::TDateTime &BackupDateTime, System::Classes::TStrings*
      BackupTables)
```

Call the BackupInfo method to retrieve information about a backup from the backup file specified by the BackupName parameter. The description for the backup is returned via the BackupDescription parameter. The date and time of the backup is returned via the BackupDateTime parameter. The tables that were backed up are returned via the BackupTables parameter.

Note
The BackupName parameter can contain a full path and file name, however when calling this method from within a remote session for a remote database you must make sure that the path is relative to the database server, not the client workstation.

TDBISAMDatabase.CloseDataSets Method

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

Call the CloseDataSets method to close all active datasets without disconnecting from the database. Ordinarily, when an application calls the Close method, all datasets are closed, and the connection to the database is dropped. Calling CloseDataSets instead of Close ensures that an application can close all active datasets without having to reconnect to the database at a later time.

TDBISAMDatabase.Commit Method

```
virtual void __fastcall Commit(bool ForceFlush = true)
```

Call the Commit method to permanently store to the database all record updates, insertions, and deletions that have occurred within the current transaction and then end the transaction. The current transaction is the last transaction started by calling the StartTransaction method. The optional ForceFlush parameter allows you to specifically indicate whether the commit should also perform an operating system flush. The default value is True.

Note
Before calling the Commit method, 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.

TDBISAMDatabase.Execute Method

```
int __fastcall Execute(const System::UnicodeString SQL,
      TDBISAMParams* Params = (TDBISAMParams*)(0x0), TDBISAMQuery*
      Query = (TDBISAMQuery*)(0x0))
```

Call the Execute method to execute an SQL statement directly. The number of rows affected is returned as the result of this method. The SQL passed to this method can be a single SQL statement or an SQL script. These SQL statements may also be parameterized.

Note
You may pass in a TDBISAMQuery component that has already been created for use with this method. However, in such a case you should be aware that many properties of the TDBISAMQuery component will be overwritten by this method in order to execute the SQL.

TDBISAMDatabase.Restore Method

```
bool __fastcall Restore(const System::UnicodeString BackupName,
      System::Classes::TStrings* BackupTables)
```

Call the Restore method to restore all tables specified by the BackupTables parameter from the backup file specified by the BackupName parameter.

The Restore method cannot be run when a transaction is currently active for the database. You can inspect the InTransaction property to determine if a transaction is currently active for the database. When the restore executes, it obtains a write lock for the entire database that prevents any sessions from performing any operation on any of the tables in the database until the restore completes.

Note
The BackupName parameter can contain a full path and file name, however when calling this method from within a remote session for a remote database you must make sure that the path is relative to the database server, not the client workstation.

TDBISAMDatabase.Rollback Method

```
virtual void __fastcall Rollback(void)
```

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

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

TDBISAMDatabase.StartTransaction Method

```
virtual void __fastcall
      StartTransaction(System::Classes::TStrings* Tables =
      (System::Classes::TStrings*)(0x0))
```

Call the StartTransaction method to begin a new transaction. Before calling the StartTransaction method, an application should check the status of the InTransaction property. If the InTransaction property is True, indicating that a transaction is already in progress, a subsequent call to StartTransaction without first calling the Commit or Rollback methods to end the current transaction will raise an exception.

The Tables parameter allows you to specify a list of table names that should be included in the transaction. This is called a restricted transaction, since it usually involves only a subset of tables in the database. If the Tables parameter is nil, the default, then the transaction will encompass the entire database.

After the StartTransaction method is called, any updates, insertions, and deletions that take place on tables that are part of the active transaction are held by DBISAM until an application calls the Commit method to save the changes or the Rollback method is to cancel them.

Note
The transaction isolation level in DBISAM is always read-committed, meaning that DBISAM will only allow sessions to view data that has been committed by another session. Any uncommitted data will be invisible until it is committed by the session.

TDBISAMDatabase.TDBISAMDatabase Method

```
__fastcall virtual TDBISAMDatabase(System::Classes::TComponent*
      AOwner)
```

Use the New operator to create an instance of a TDBISAMDatabase component using the constructor.

TDBISAMDatabase.ValidateName Method

```
void __fastcall ValidateName(const System::UnicodeString Name)
```

Call the ValidateName method to prevent duplicate access to a TDBISAMDatabase component from within a single TDBISAMSession component. The Name parameter contains the DatabaseName of the TDBISAMDatabase component to test. If the TDBISAMDatabase component is already open, the ValidateName method raises an exception. If the TDBISAMDatabase component is not open, the procedure returns, and the application continues processing.

Note
Most applications should not need to call this method directly. It is called automatically each time a TDBISAMDatabase component is opened.

TDBISAMDatabase.OnBackupLog Event

```
__property TLogEvent OnBackupLog
```

The OnBackupLog event is fired when an application calls the Backup method and the backup operation needs to report a backup log message to the application.

TDBISAMDatabase.OnBackupProgress Event

```
__property TSteppedProgressEvent OnBackupProgress
```

The OnBackupProgress event is fired when an application calls the Backup method and the backup operation needs to report a backup progress message to the application.

Note
The number of times that this event fires is controlled by the TDBISAMSession ProgressSteps property of the current session.

TDBISAMDatabase.OnCommit Event

```
__property System::Classes::TNotifyEvent OnCommit
```

The OnCommit event is fired after an application calls the Commit method and the commit operation succeeds.

TDBISAMDatabase.OnRestoreLog Event

```
__property TLogEvent OnRestoreLog
```

The OnRestoreLog event is fired when an application calls the Restore method and the restore operation needs to report a restore log message to the application.

TDBISAMDatabase.OnRestoreProgress Event

```
__property TSteppedProgressEvent OnRestoreProgress
```

The OnRestoreProgress event is fired when an application calls the Restore method and the restore operation needs to report a restore progress message to the application.

Note
The number of times that this event fires is controlled by the TDBISAMSession ProgressSteps property of the current session.

TDBISAMDatabase.OnRollback Event

```
__property System::Classes::TNotifyEvent OnRollback
```

The OnRollback event is fired after an application calls the Rollback method and the rollback operation succeeds.

TDBISAMDatabase.OnStartTransaction Event

```
__property System::Classes::TNotifyEvent OnStartTransaction
```

The OnStartTransaction event is fired after an application calls the StartTransaction method, but before the actual transaction is started.

5.5 TDBISAMDataSet Component

Header File: dbisamtb

Inherits From TDBISAMBaseDataSet

The TDBISAMDataSet component is a dataset component that defines DBISAM-specific functionality for a dataset. Applications never use TDBISAMDataSet components directly. Instead they use the descendants of TDBISAMDataSet, the TDBISAMQuery and TDBISAMTable components, which inherit its database-related properties and methods.

Properties	Methods	Events
AutoDisplayLabels	ApplyCachedUpdates	OnCachedUpdateError
CachedUpdatesModified	BeginCachedUpdates	OnUpdateRecord
CachingUpdates	BookmarkValid	
CopyOnAppend	Cancel	
FilterExecutionTime	CancelCachedUpdates	
FilterOptimizeLevel	CompareBookmarks	
FilterRecordCount	CreateBlobStream	
Handle	ExportTable	
KeySize	FlushBuffers	
LocalReadSize	GetBlobFieldData	
RecordHash	GetCurrentRecord	
RecordID	GetFieldClass	
RemoteReadSize	GetFieldData	
StoreActive	GetIndexInfo	
UpdateObject	GetNextRecords	
	GetPriorRecords	
	ImportTable	
	IsSequenced	
	LoadFromStream	
	Locate	
	Lookup	
	Post	
	SaveToStream	
	TDBISAMDataSet	

TDBISAMDataSet.AutoDisplayLabels Property

```
__property bool AutoDisplayLabels
```

Use the AutoDisplayLabels property to specify whether the descriptions for each field in the dataset should be automatically populated as the DisplayLabel property of each TField component defined for this TDBISAMDataSet component. This helps alleviate the hassle of constantly having to manually update multiple instances of a TField component associated with multiple TDBISAMTable or TDBISAMQuery components when you wish to change the "pretty" version of a field name. Since the TDBGrid component uses the DisplayLabel property of a TField component automatically, this property is very useful when data will be displayed in a TDBGrid component.

Note
This property is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.CachedUpdatesModified Property

```
__property bool CachedUpdatesModified
```

TDBISAMDataSet.CachingUpdates Property

```
__property bool CachingUpdates
```

Use the CachingUpdates property to determine whether updates are being cached.

TDBISAMDataSet.CopyOnAppend Property

```
__property bool CopyOnAppend
```

Use the CopyOnAppend property to control whether the current or last record's contents should be copied automatically to any newly inserted or appended records.

Note
Using the Append method will cause the last record to be copied, not the current record. If you wish to copy the current record's contents then you should use the Insert method. Also, this property is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.FilterExecutionTime Property

```
__property double FilterExecutionTime
```

Use the FilterExecutionTime property to determine how long the current filter took to execute in seconds.

TDBISAMDataSet.FilterOptimizeLevel Property

```
__property TFilterOptimizeLevel FilterOptimizeLevel
```

Use the FilterOptimizeLevel property to determine the optimization level of the current expression filter specified via the Filter property.

You may examine this property after the Filtered property is set to True.

Note
This property is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.FilterRecordCount Property

```
__property int FilterRecordCount
```

Use the FilterRecordCount property to determine the total number of records, including active filters (or WHERE clauses with live query result sets), when the TDBISAMEngine FilterRecordCounts property is set to False. If the TDBISAMEngine FilterRecordCounts property is set to True, then the FilterRecordCount property always matches that of the RecordCount property.

Note
This property is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.Handle Property

```
__property Dbisamen::TDBISAMCursor* Handle
```

The Handle property is for internal use only and is not useful to the application developer using DBISAM.

TDBISAMDataSet.KeySize Property

```
__property System::Word KeySize
```

The KeySize property specifies the size, in bytes, of a key in the active index. KeySize varies depending on the number and type of fields that make up the active index. The key size is for internal use only and is not useful to the application developer using DBISAM.

Note
This property is only used in the context of a descendant TDBISAMTable component.

TDBISAMDataSet.LocalReadSize Property

```
__property int LocalReadSize
```

Use the LocalReadSize property to specify how many records should be read at once whenever a local session needs to read records from disk. This property is most useful when performing a sequential navigation of a large table or query result set in a network sharing environment. However, you should be careful to not set this property to too high of a value since doing so can result in excessive memory consumption and network traffic. This is especially true when the access to the table or query result set is mostly random and not sequential.

The actual number of records read is dependent upon the value of this property combined with an internal intelligent read-ahead calculation performed by the engine, with the greater of the two values being used. This means that if you increase this value from the default value of 1, then you may be causing the engine to read more records than it normally would. Therefore, it is important that you also increase the record buffering settings of the engine to avoid forcing the engine to spend excessive amounts of time ejecting records from the cache. You can do so by modifying these properties in the TDBISAMEngine component:

MaxTableDataBufferCount
MaxTableDataBufferSize

Note
This property is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.RecordHash Property

```
__property System::UnicodeString RecordHash
```

The RecordHash property indicates the MD5 hash value of the current record in the form of a string.

Note
This property is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.RecordID Property

```
__property int RecordID
```

The RecordID property indicates the inviolate record ID of the current record.

Note
This property is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.RemoteReadSize Property

```
__property int RemoteReadSize
```

Use the RemoteReadSize property to specify how many records should be read at once whenever a remote session needs to read records from a database server. This property is most useful when performing a sequential navigation of a large remote table or query result set on a database server. You should be careful to not set this property to too high of a value since doing so can result in excessive memory consumption and network traffic. This is especially true when the access to a remote table or query result set is mostly random and not sequential.

Note
This property is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.StoreActive Property

```
__property bool StoreActive
```

Use the StoreActive property to determine if the dataset should store the current value of its Active property in the owner form or data module. The default value for this property is True.

TDBISAMDataSet.UpdateObject Property

```
__property TDBISAMDataSetUpdateObject* UpdateObject
```

Use the UpdateObject property to specify a TDBISAMUpdateSQL component that will be used to apply any updates from a TClientDataSet component via the IProvider support in DBISAM.

TDBISAMDataSet.ApplyCachedUpdates Method

```
void __fastcall ApplyCachedUpdates(void)
```

Use the ApplyCachedUpdates method to begin the process of apply any inserts, updates, or deletes that were cached to the source table or query result set. If there are any errors during this process, you can use an OnCachedUpdateError event handler to handle the errors and reconcile them so that the application of the updates can complete successfully.

Note
You should always wrap the ApplyCachedUpdates method with a transaction. This allows the application of the updates to either fail or succeed as a single atomic unit of work.

TDBISAMDataSet.BeginCachedUpdates Method

```
void __fastcall BeginCachedUpdates(void)
```

Use the BeginCachedUpdates method to copy all records to a temporary table that will be used for caching all inserts, updates, and deletes until the cached updates are applied using the ApplyCachedUpdates method or cancelled using the CancelCachedUpdates method.

TDBISAMDataSet.BookmarkValid Method

```
virtual bool __fastcall
      BookmarkValid(System::DynamicArray<System::Byte> Bookmark)
```

Use the BookmarkValid method to determine if a specifed bookmark is currently assigned a valid bookmark value. Bookmark specifies the bookmark to test. BookmarkValid returns True if a bookmark is valid. Otherwise, it returns False.

Note
This method is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.Cancel Method

```
virtual void __fastcall Cancel(void)
```

This method is defined in the TDataSet Cancel topic.

TDBISAMDataSet.CancelCachedUpdates Method

```
void __fastcall CancelCachedUpdates(void)
```

Use the CancelCachedUpdates method to discard any cached updates and return the source table or query result set to its original state.

TDBISAMDataSet.CompareBookmarks Method

```
virtual int __fastcall
      CompareBookmarks(System::DynamicArray<System::Byte> Bookmark1,
      System::DynamicArray<System::Byte> Bookmark2)
```

Use the CompareBookmarks method to determine if two bookmarks are identical or not. Bookmark1 and Bookmark2 are the bookmarks to compare. If the bookmarks differ, CompareBookmarks returns 1. If the Bookmarks are identical, or both bookmarks are nil, CompareBookmarks returns 0.

Note
This method is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.CreateBlobStream Method

```
virtual System::Classes::TStream* __fastcall
      CreateBlobStream(Data::Db::TField* Field,
      Data::Db::TBlobStreamMode Mode)
```

Use the CreateBlobStream method to create a BLOB 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. The Mode parameter specifies whether the stream will be used for reading, writing, or updating the contents of the field.

Note
This method is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.ExportTable Method

```
virtual void __fastcall ExportTable(const System::UnicodeString
      ExportToFile, System::WideChar Delimiter, bool WriteHeaders =
      false, System::Classes::TStrings* FieldsToExport =
      (System::Classes::TStrings*)(0x0), const System::UnicodeString
      DateFormat = L"yyyy-mm-dd", const System::UnicodeString
      TimeFormat = L"hh:mm:ss.zzz ampm", System::WideChar DecSeparator
      = (System::WideChar)(0x2e)) = 0
```

Call the ExportTable method to export data from the dataset into a delimited text file. The dataset may be open or closed when executing this method. If the dataset is open, then this method will respect any active filters or ranges on the dataset when copying the data to the delimited text file.

The FileToExport parameter indicates the name (including path) of the delimited text file to create when exporting the dataset contents.

The Delimiter parameter indicates the char to be used as the delimiter in the text file being created during the export.

The WriteHeaders parameter indicates whether to write the field names being exported to the first line of the text file being created during the export.

The FieldsToExport parameter indicates the names of the fields that should be populated with data from the dataset when exporting. This is useful for situations where the structure of the outgoing data needs to be different from that of the source dataset.

The DateFormat parameter indicates the date formatting to be used for any date or timestamp (date and time) fields in the exported text file. The rules for the date format specification are the same as with Delphi, Kylix, and C++Builder date formats. The only restriction is that you must include a date separator between the year (y), month (m), and day (d) placeholders. When exporting timestamp fields the formatting is assumed to be DateFormat, space character, and then TimeFormat. All formatting is case-insensitive and the default date format is "yyyy-mm-dd".

The TimeFormat parameter indicates the time formatting to be used for any time or timestamp (date and time) fields in the exported text file. The rules for the time format specification are the same as with Delphi, Kylix, and C++Builder time formats. The only restriction is that you must include a time separator between the hours (h), minutes (m), and seconds (s) placeholders. Also, any milliseconds formatting must use a period "." as the separator between the seconds and the milliseconds placeholders. Finally, when specifying the "ampm" switch for 12-hour clock format, do not use a forward slash between the "am" and "pm" placeholders. When exporting timestamp fields the formatting is assumed to be DateFormat, space character, and then TimeFormat. All formatting is case-insensitive and the default time format is "hh:mm:ss.zzz ampm".

The DecSeparator parameter indicates the decimal separator to be used for any number fields (Float or BCD) in the exported text file. DBISAM does not support the use of thousands separators in exported number field text data, only decimal separators. The default decimal separator is ".".

TDBISAMDataSet.FlushBuffers Method

```
void __fastcall FlushBuffers(void)
```

Use the FlushBuffers method to flush data to disk. If the table or query result set is opened exclusively, then the FlushBuffers method flushes all cached writes to disk and proceeds to instruct the operating system to flush all writes to disk also. If the table or query result set is opened shared, then FlushBuffers only instructs the operating system to flush all writes to disk since shared datasets do not cache any writes.

Note
This method is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.GetBlobFieldData Method

```
virtual int __fastcall GetBlobFieldData(int FieldNo,
      System::DynamicArray<System::Byte> &Buffer)
```

Use the GetBlobFieldData method to retrieve the contents of a BLOB field directly into an application-allocated buffer.

Note
This method is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.GetCurrentRecord Method

```
virtual bool __fastcall GetCurrentRecord(System::PByte Buffer)
```

This method is only used internally by DBISAM and should be ignored by application developers.

TDBISAMDataSet.GetFieldClass Method

```
virtual Data::Db::TFieldClass __fastcall
      GetFieldClass(Data::Db::TFieldType FieldType)
```

This method is only used internally by DBISAM and should be ignored by application developers.

TDBISAMDataSet.GetFieldData Method

```
virtual bool __fastcall GetFieldData(Data::Db::TField* Field,
      System::DynamicArray<System::Byte> &Buffer)

virtual bool __fastcall GetFieldData(int FieldNo,
      System::DynamicArray<System::Byte> &Buffer)

virtual bool __fastcall GetFieldData(Data::Db::TField* Field,
      void * Buffer)

virtual bool __fastcall GetFieldData(int FieldNo, void * Buffer)
```

This method is only used internally by DBISAM and should be ignored by application developers.

TDBISAMDataSet.GetIndexInfo Method

```
void __fastcall GetIndexInfo(void)
```

This method is only used internally by DBISAM and should be ignored by application developers.

TDBISAMDataSet.GetNextRecords Method

```
virtual int __fastcall GetNextRecords(void)
```

This method is only used internally by DBISAM and should be ignored by application developers.

TDBISAMDataSet.GetPriorRecords Method

```
virtual int __fastcall GetPriorRecords(void)
```

This method is only used internally by DBISAM and should be ignored by application developers.

TDBISAMDataSet.ImportTable Method

```
virtual void __fastcall ImportTable(const System::UnicodeString
      FileToImport, System::WideChar Delimiter, bool ReadHeaders =
      false, System::Classes::TStrings* FieldsToImport =
      (System::Classes::TStrings*)(0x0), const System::UnicodeString
      DateFormat = L"yyyy-mm-dd", const System::UnicodeString
      TimeFormat = L"hh:mm:ss.zzz ampm", System::WideChar DecSeparator
      = (System::WideChar)(0x2e)) = 0
```

Call the ImportTable method to import data into the dataset from a delimited text file. The dataset may be open or closed when executing this method.

The FileToImport parameter indicates the name (including path) of the delimited text file to import into the dataset.

The Delimiter parameter indicates the char used as the delimiter in the text file being imported into the dataset.

The ReadHeaders parameter indicates whether to read the field names to be imported directly from the first line of the text file being imported.

The FieldsToImport parameter indicates the names of the fields that should be populated with data from the incoming delimited text file. This is useful for situations where the structure of the incoming data does not match that of the dataset.

The DateFormat parameter indicates the date formatting to be used for interpreting any incoming date or timestamp (date and time) fields in the imported text file. The rules for the date format specification are the same as with Delphi, Kylix, and C++Builder date formats. The only restriction is that you must include a date separator between the year (y), month (m), and day (d) placeholders. When importing timestamp fields the formatting is assumed to be DateFormat, space character, and then TimeFormat. All formatting is case-insensitive and the default date format is "yyyy-mm-dd".

The TimeFormat parameter indicates the time formatting to be used for interpreting any incoming time or timestamp (date and time) fields in the imported text file. The rules for the time format specification are the same as with Delphi, Kylix, and C++Builder time formats. The only restriction is that you must include a time separator between the hours (h), minutes (m), and seconds (s) placeholders. Also, any milliseconds formatting must use a period "." as the separator between the seconds and the milliseconds placeholders. Finally, when specifying the "ampm" switch for 12-hour clock format, do not use a forward slash between the "am" and "pm" placeholders. When importing timestamp fields the formatting is assumed to be DateFormat, space character, and then TimeFormat. All formatting is case-insensitive and the default time format is "hh:mm:ss.zzz ampm".

The DecSeparator parameter indicates the decimal separator to be used for interpreting any incoming number fields (Float or BCD) in the imported text file. DBISAM does not support the use of thousands separators in incoming number field text data, only decimal separators. The default decimal separator is ".".

TDBISAMDataSet.IsSequenced Method

```
virtual bool __fastcall IsSequenced(void)
```

Use the IsSequenced method to determine whether records can be located by logical record numbers. When the IsSequenced method returns True, you can navigate directly to a specific record by setting the RecNo property. If the IsSequenced method returns False, the only way to navigate to a specific record is to start at the beginning and count records.

Note
This method is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.LoadFromStream Method

```
virtual void __fastcall LoadFromStream(System::Classes::TStream*
      SourceStream) = 0
```

Call the LoadFromStream method to load the contents of the dataset from a stream containing data previously created using the SaveToStream method.

TDBISAMDataSet.Locate Method

```
virtual bool __fastcall Locate(const System::UnicodeString
      KeyFields, const System::Variant &KeyValues,
      Data::Db::TLocateOptions Options)
```

Use the Locate method to search for a specified record and makes that record the current record. KeyFields is a string containing a semicolon-delimited list of field names on which to search. KeyValues is a variant that specifies 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. 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 KeyFields does not include any string fields, Options is ignored.

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

Note
This method is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.Lookup Method

```
virtual System::Variant __fastcall Lookup(const
      System::UnicodeString KeyFields, const System::Variant
      &KeyValues, const System::UnicodeString ResultFields)
```

Use the Lookup method to retrieve values for specified fields from a record that matches search criteria. KeyFields is a string containing a semicolon-delimited list of field names on which to search. KeyValues is a variant array containing the values to match in the key fields. To specify multiple search values, pass KeyValues as a variant array as an argument, or construct a variant array on the fly using the VarArrayOf routine. ResultFields is a string containing a semicolon-delimited list of field names whose values should be returned from the matching record.

Lookup returns a variant array containing the values from the fields specified in ResultFields. 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.

Note
This method is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDataSet.Post Method

```
virtual void __fastcall Post(void)
```

This method is defined in the TDataSet Post topic.

TDBISAMDataSet.SaveToStream Method

```
virtual void __fastcall SaveToStream(System::Classes::TStream*
      DestStream) = 0
```

Call the SaveToStream method to save the contents of the dataset to a stream. You can then use LoadFromStream method to load the data from the stream using another TDBISAMTable or TDBISAMQuery component.

Note
Do not use this method with very large datasets. It is recommended that you do not use it with datasets over a few megs in size.

TDBISAMDataSet.TDBISAMDataSet Method

```
__fastcall virtual TDBISAMDataSet(System::Classes::TComponent*
      AOwner)
```

Use the New operator to create an instance of a TDBISAMDataSet component using the constructor. However, you should not use this constructor to create an instance of this component directly. Instead you should create an instance of a TDBISAMTable or TDBISAMQuery component, which both descend from this component and provide a published interface for use at design-time.

TDBISAMDataSet.OnCachedUpdateError Event

```
__property TCachedUpdateErrorEvent OnCachedUpdateError
```

The OnCachedUpdateError event is fired when there is an error during the application of a cached update. You may use the CurrentRecord parameter to examine and/or modify the record being applied. The E paramater contains the exception that was raised. The UpdateType parameter indicates what type of operation was being applied. Finally, the Action parameter allows you to specify what action you would like DBISAM to take in response to any adjustments that may have been made to the CurrentRecord object.

TDBISAMDataSet.OnUpdateRecord Event

```
__property Data::Db::TUpdateRecordEvent OnUpdateRecord
```

The OnUpdateRecord event is fired when the IProvider support in DBISAM is attempting to apply an update from a TClientDataSet component. Write an event handler for this event to intercept an update before it is applied automatically by DBISAM. This will allow you to provide custom processing for situations where the standard update processing is not sufficient.

5.6 TDBISAMDataSetUpdateObject Component

Header File: dbisamtb

Inherits From Classes

The TDBISAMDataSetUpdate component is an abstract component that is implemented by the TDBISAMUpdateSQL component. Normally, only developers interested in creating their own custom update components would use the TDBISAMDataSetUpdateObject.

Properties	Methods	Events
	TDBISAMDataSetUpdateObject	

TDBISAMDataSetUpdateObject.TDBISAMDataSetUpdateObject Method

```
inline __fastcall virtual
      TDBISAMDataSetUpdateObject(System::Classes::TComponent* AOwner)
      : System::Classes::TComponent(AOwner) { }
```

Use the New operator to create an instance of a TDBISAMDataSetUpdateObject component using the constructor. However, you should not use this constructor to create an instance of this component directly. Instead you should create an instance of a TDBISAMUpdateSQL component, which descends from this component and provides a published interface for use at design-time.

5.7 TDBISAMDBDataSet Component

Header File: dbisamtb

Inherits From TDBISAMDataSet

The TDBISAMDBDataSet component is a dataset component that defines database-related connectivity properties and methods for a DBISAM dataset. Applications never use TDBISAMDBDataSet components directly. Instead they use the descendants of TDBISAMDBDataSet, the TDBISAMQuery and TDBISAMTable components, which inherit its database-related properties and methods.

Properties	Methods	Events
Database	CloseDatabase	
DatabaseName	OpenDatabase	
DBHandle	TDBISAMDBDataSet	
DBSession		
SessionName		

TDBISAMDBDataSet.Database Property

```
__property TDBISAMDatabase* Database
```

Use the Database property to access the properties, events, and methods of the TDBISAMDatabase component linked to this TDBISAMDBDataSet component. The Database property is read-only and is automatically set when the database specified by the DatabaseName property is opened.

Note
This property is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDBDataSet.DatabaseName Property

```
__property System::UnicodeString DatabaseName
```

Use the DatabaseName property to specify the name of the TDBISAMDatabase component to link to this TDBISAMDBDataSet component. The DatabaseName property should match the DatabaseName property of an existing TDBISAMDatabase component or should specify a valid local path name, for local sessions, or a remote database name, for remote sessions.

Note
Attempting to set the DatabaseName property when the TDBISAMDBDataSet component is open (Active=True) will raise an exception. Also, this property is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDBDataSet.DBHandle Property

```
__property Dbisamen::TDBISAMDatabaseManager* DBHandle
```

The DBHandle property is for internal use only and is not useful to the application developer using DBISAM.

TDBISAMDBDataSet.DBSession Property

```
__property TDBISAMSession* DBSession
```

Use the DBSession property to determine the TDBISAMSession component with which this TDBISAMDBDataSet component is linked. By default, the TDBISAMDBDataset component is linked with the default TDBISAMSession component, Session, that is automatically created by DBISAM.

Note
This property is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDBDataSet.SessionName Property

```
__property System::UnicodeString SessionName
```

Use the SessionName property to specify the TDBISAMSession component to link to this TDBISAMDBDataSet component. If the SessionName property is blank, the TDBISAMDBDataSet component is automatically linked to the default TDBISAMSession component, Session. To link the TDBISAMDBDataset component with a different TDBISAMSession component, the SessionName property must match the SessionName property of an existing TDBISAMSession component.

Note
This property is only used in the context of the descendant TDBISAMTable and TDBISAMQuery components.

TDBISAMDBDataSet.CloseDatabase Method

```
void __fastcall CloseDatabase(TDBISAMDatabase* Database)
```

The CloseDatabase method is just a local version of the TDBISAMSession CloseDatabase method for the TDBISAMSession that the TDBISAMDBDataSet is linked to via its SessionName property.

TDBISAMDBDataSet.OpenDatabase Method

```
TDBISAMDatabase* __fastcall OpenDatabase(void)
```

The OpenDatabase method is just a local version of the TDBISAMSession OpenDatabase method for the TDBISAMSession that the TDBISAMDBDataSet is linked to via its SessionName property.

TDBISAMDBDataSet.TDBISAMDBDataSet Method

```
inline __fastcall virtual
      TDBISAMDBDataSet(System::Classes::TComponent* AOwner) :
      TDBISAMDataSet(AOwner) { }
```

You should not use this constructor to create an instance of this component directly. Instead you should create an instance of a TDBISAMTable or TDBISAMTable component, which both descend from this component and provide a published interface for use at design-time.

5.8 TDBISAMEngine Component

Header File: dbisamtb

Inherits From Classes

Use the TDBISAMEngine component to manage the DBISAM engine from within an application. The DBISAM engine can behave as either a local, or client, engine or as a database server engine.

A default TDBISAMEngine component is created automatically when the application is started and can be referenced via the global Engine function in the dbisamtb unit (Delphi) and dbisamtb header file (C++).

Note
Use of any of the properties, methods, and events in the TDBISAMEngine component is completely optional and not required for proper use of DBISAM. However, the TDBISAMEngine component can be a very powerful tool for customizing DBISAM, especially when it is running as a database server. Please see the Configuring and Starting the Server and Customizing the Engine topics for more information.

Properties	Methods	Events
Active	AddServerDatabase	AfterDeleteTrigger
CreateTempTablesInDatabase	AddServerDatabaseUser	AfterInsertTrigger
EngineSignature	AddServerEvent	AfterUpdateTrigger
EngineType	AddServerProcedure	BeforeDeleteTrigger
EngineVersion	AddServerProcedureUser	BeforeInsertTrigger
FilterRecordCounts	AddServerUser	BeforeUpdateTrigger
Functions	AnsiStrToBoolean	CommitTrigger
LockFileName	AnsiStrToCurr	OnCompress
MaxTableBlobBufferCount	AnsiStrToDate	OnCryptoInit
MaxTableBlobBufferSize	AnsiStrToDateTime	OnCryptoReset
MaxTableDataBufferCount	AnsiStrToFloat	OnCustomFunction
MaxTableDataBufferSize	AnsiStrToTime	OnDecompress
MaxTableIndexBufferCount	BooleanToAnsiStr	OnDecryptBlock
MaxTableIndexBufferSize	BuildWordList	OnDeleteError
ServerAdminAddress	ConvertIDToLocaleConstant	OnEncryptBlock
ServerAdminPort	ConvertLocaleConstantToID	OnInsertError
ServerAdminThreadCacheSize	CurrToAnsiStr	OnServerConnect
ServerConfigFileName	DateTimeToAnsiStr	OnServerDisconnect
ServerConfigPassword	DateToAnsiStr	OnServerLogCount

ServerDescription	DeleteServerDatabase	OnServerLogEvent
ServerEncryptedOnly	DeleteServerDatabaseUser	OnServerLogin
ServerEncryptionPassword	DeleteServerEvent	OnServerLogout
ServerLicensedConnections	DeleteServerProcedure	OnServerLogRecord
ServerMainAddress	DeleteServerProcedureUser	OnServerProcedure
ServerMainPort	DeleteServerUser	OnServerReconnect
ServerMainThreadCacheSize	DisconnectServerSession	OnServerScheduledEvent
ServerName	FindSession	OnServerStart
SessionCount	FloatToAnsiStr	OnServerStop
SessionList	GetDefaultTextIndexParams	OnShutdown
Sessions	GetLocaleNames	OnStartup
StoreActive	GetServerConfig	OnTextIndexFilter
TableBlobBackupExtension	GetServerConnectedSessionCount	OnTextIndexTokenFilter
TableBlobExtension	GetServerDatabase	OnUpdateError
TableBlobTempExtension	GetServerDatabaseNames	RecordLockTrigger
TableBlobUpgradeExtension	GetServerDatabaseUser	RecordUnlockTrigger
TableDataBackupExtension	GetServerDatabaseUserNames	RollbackTrigger
TableDataExtension	GetServerEvent	SQLTrigger
TableDataTempExtension	GetServerEventNames	StartTransactionTrigger
TableDataUpgradeExtension	GetServerLogCount	
TableFilterIndexThreshhold	GetServerLogRecord	
TableIndexBackupExtension	GetServerMemoryUsage	
TableIndexExtension	GetServerProcedure	
TableIndexTempExtension	GetServerProcedureNames	
TableIndexUpgradeExtension	GetServerProcedureUser	
TableMaxReadLockCount	GetServerProcedureUserNames	
TableReadLockTimeout	GetServerSessionCount	
TableTransLockTimeout	GetServerSessionInfo	
TableWriteLockTimeout	GetServerUpTime	
	GetServerUser	
	GetServerUserNames	
	GetServerUTCDateTime	
	GetSessionNames	
	IsValidLocale	
	IsValidLocaleConstant	
	ModifyServerConfig	

	ModifyServerDatabase	
	ModifyServerDatabaseUser	
	ModifyServerEvent	
	ModifyServerProcedure	
	ModifyServerProcedureUser	
	ModifyServerUser	
	ModifyServerUserPassword	
	OpenSession	
	QuotedSQLStr	
	RemoveServerSession	
	StartAdminServer	
	StartMainServer	
	StopAdminServer	
	StopMainServer	
	TDBISAMEngine	
	TimeToAnsiStr	

TDBISAMEngine.Active Property

```
__property bool Active
```

Use the Active property to specify whether or not the engine is active. Setting Active to True starts the engine.

If the EngineType property is set to etServer, then DBISAM will attempt to start the engine as a database server using the:

ServerConfigFileName
ServerConfigPassword
ServerName
ServerDescription
ServerEncryptedOnly
ServerEncryptionPassword
ServerMainAddress
ServerMainPort
ServerMainThreadCacheSize
ServerAdminAddress
ServerAdminPort
ServerAdminThreadCacheSize

properties. Also, the OnServerStart event will be triggered.

If the EngineType property is set to etClient (the default), then DBISAM will simply start the engine.

Setting Active to False closes any open datasets, disconnects active database connections, and stops all active sessions.

TDBISAMEngine.CreateTempTablesInDatabase Property

```
__property bool CreateTempTablesInDatabase
```

Use the CreateTempTablesInDatabase property to control the location of any temporary tables created by the TDBISAMTable AlterTable, OptimizeTable, or UpgradeTable methods, or their SQL counterparts, the ALTER TABLE statement, the OPTIMIZE TABLE statement, or the UPGRADE TABLE Statement. The default value of this property is False, which means that any temporary tables are stored in the location specified by the TDBISAMSession PrivateDir property. If this property is set to True, then any temporary tables will be created in the current database directory specified by the TDBISAMTable or TDBISAMQuery DatabaseName property, or specified by the TDBISAMDatabase Directory property. This property only applies to the above operations, and does not apply to the temporary tables created by DBISAM for canned query result sets.

Note
Setting this property to True will help avoid any user rights issues that may occur with Windows XP and higher when altering or optimizing tables. Under these operating systems, the rights assigned to any temporary tables created in local directories are usually very restrictive with respect to sharing the table with other users. When the temporary table is copied back to the main database directory once the table alteration, optimization, or upgrade is complete, the table is copied with these restrictive rights. This can make the table inaccessible to other users that need to open, read, or write to the table.

TDBISAMEngine.EngineSignature Property

```
__property System::UnicodeString EngineSignature
```

Use the EngineSignature property to specify the signature to be used by the engine when accessing or creating tables, backup files, or streams as well as any communications between a remote session and a database server. The default value of the EngineSignature property is "DBISAM_SIG" and should not be changed unless you are sure of the consequences. Using a custom value for the EngineSignature property will prevent any other application that uses DBISAM from accessing any tables, backup files, or streams created with the custom signature, as well as accessing a database server using the custom signature.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.EngineType Property

```
__property TEngineType EngineType
```

Use the EngineType property to specify whether the engine should behave as a local, client engine (the default) or as a database server engine. DBISAM only allows one instance of the TDBISAMEngine component per application, which means that an application can only behave as a local, client application, or as a database server application, but not both.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.EngineVersion Property

```
__property System::UnicodeString EngineVersion
```

Indicates the current version of DBISAM being used. This property is read-only, but published so that it is visible in the Object Inspector in Delphi, Kylix, and C++Builder.

TDBISAMEngine.FilterRecordCounts Property

```
__property bool FilterRecordCounts
```

Use the FilterRecordCounts property to specify how record counts are returned for filtered datasets and live query result sets. The default value of this property is True, which means the RecordCount property of the TDBISAMTable and TDBISAMQuery components will always take into account any active filters (or WHERE clauses with live query result sets) when showing the record count of the dataset. If the FilterRecordCounts property is set to False, the RecordCount property of the TDBISAMTable and TDBISAMQuery components will always show the total record count of the entire dataset or active range (if a range is set) only and will not take any active filters (or WHERE clauses with live query result sets) into account. To get the record count including any active filters, use the FilterRecordCount property of the TDBISAMTable and TDBISAMQuery components. This property always shows the accurate record count, regardless of the current setting of the TDBISAMEngine FilterRecordCounts property.

Setting the TDBISAMEngine FilterRecordCounts property to False may be desirable for some applications since it allows for more accurate positioning of the scroll bar in a TDBGrid or similar multi-row, data-aware components.

TDBISAMEngine.Functions Property

```
__property TDBISAMFunctions* Functions
```

Use the Functions property to define custom functions for use with filter expressions and SQL statements.

Note
Adding a custom function while the engine is active will result in the engine triggering an exception. You should define all custom functions before activating the engine.

TDBISAMEngine.LockFileName Property

```
__property System::UnicodeString LockFileName
```

Use the LockFileName property to specify the lock file name used by DBISAM for placing table and transaction locks for each physical database directory.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.MaxTableBlobBufferCount Property

```
__property int MaxTableBlobBufferCount
```

Use the MaxTableBlobBufferCount property to specify the maximum number of BLOB block buffers to allow in the buffer cache per physical table. This property only applies to locally-opened tables and is ignored for tables opened on a remote database server. Increasing this value will help increase performance at the cost of increased memory consumption per table.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.MaxTableBlobBufferSize Property

```
__property int MaxTableBlobBufferSize
```

Use the MaxTableBlobBufferSize property to specify the total amount of memory to use for BLOB block buffers in the buffer cache per physical table. This property only applies to locally-opened tables and is ignored for tables opened on a remote database server. Increasing this value will help increase performance at the cost of increased memory consumption per table.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.MaxTableDataBufferCount Property

```
__property int MaxTableDataBufferCount
```

Use the MaxTableDataBufferCount property to specify the maximum number of record buffers to allow in the buffer cache per physical table. This property only applies to locally-opened tables and is ignored for tables opened on a remote database server. Increasing this value will help increase performance at the cost of increased memory consumption per table.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.MaxTableDataBufferSize Property

```
__property int MaxTableDataBufferSize
```

Use the MaxTableDataBufferSize property to specify the total amount of memory to use for record buffers in the buffer cache per physical table. This property only applies to locally-opened tables and is ignored for tables opened on a remote database server. Increasing this value will help increase performance at the cost of increased memory consumption per table.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.MaxTableIndexBufferCount Property

```
__property int MaxTableIndexBufferCount
```

Use the MaxTableIndexBufferCount property to specify the maximum number of index page buffers to allow in the buffer cache per physical table. This property only applies to locally-opened tables and is ignored for tables opened on a remote database server. Increasing this value will help increase performance at the cost of increased memory consumption per table.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.MaxTableIndexBufferSize Property

```
__property int MaxTableIndexBufferSize
```

Use the MaxTableIndexBufferSize property to specify the total amount of memory to use for index page buffers in the buffer cache per physical table. This property only applies to locally-opened tables and is ignored for tables opened on a remote database server. Increasing this value will help increase performance at the cost of increased memory consumption per table.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.ServerAdminAddress Property

```
__property System::UnicodeString ServerAdminAddress
```

Use the ServerAdminAddress property to specify the IP address that the database server should listen on for administrative connections when the EngineType property is set to etServer. A blank value (the default) indicates that the database server should listen on all available IP addresses from the operating system.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.ServerAdminPort Property

```
__property int ServerAdminPort
```

Use the ServerAdminPort property to specify the port that the database server should listen on for administrative connections when the EngineType property is set to etServer. The default value is 12006.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.ServerAdminThreadCacheSize Property

```
__property int ServerAdminThreadCacheSize
```

Use the ServerAdminThreadCacheSize property to specify the total number of threads that should be cached by the database server for administrative connections when the EngineType property is set to etServer. The default value is 1. Caching threads helps improve connection times by eliminating the need to constantly create and destroy threads as remote sessions connect to and disconnect from the database server.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.ServerConfigFileName Property

```
__property System::UnicodeString ServerConfigFileName
```

Use the ServerConfigFileName property to specify the name of the configuration file the database server should use for storing all configuration information when the EngineType property is set to etServer. The default value is "dbsrvr". You should not specify a file extension in this file name since DBISAM always uses a default file extension of ".scf" for database server configuration files.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.ServerConfigPassword Property

```
__property System::UnicodeString ServerConfigPassword
```

Use the ServerConfigPassword property to specify the password used to encrypt the the configuration file the database server uses for storing all configuration information when the EngineType property is set to etServer. The default value is "elevatesoft". The name of the configuration file itself is specified in the ServerConfigFileName property.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.ServerDescription Property

```
__property System::UnicodeString ServerDescription
```

Use the ServerDescription property to specify the description of the database server when the EngineType property is set to etServer. The default value is "DBISAM Database Server". This description is used to describe the database server when a remote session asks for the description using the TDBISAMSession GetRemoteServerDescription method.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.ServerEncryptedOnly Property

```
__property bool ServerEncryptedOnly
```

Use the ServerEncryptedOnly property to specify that the database server should only accept encrypted regular data connections when the EngineType property is set to etServer. The default value is False, however administrative connections to the database server are always required to be encrypted.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.ServerEncryptionPassword Property

```
__property System::UnicodeString ServerEncryptionPassword
```

Use the ServerEncryptedPassword property to specify the password the database server should use for communicating with encrypted administrative or regular data connnections when the EngineType property is set to etServer. The default value is "elevatesoft". Administrative connections to the database server are always required to be encrypted.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.ServerLicensedConnections Property

```
__property System::Word ServerLicensedConnections
```

Use the ServerLicensedConnections property to specify the maximum number of licensed connections allowed for the database server.

Note
This property overrides any maximum connection settings that may already be present in the database server configuration. Also, this property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.ServerMainAddress Property

```
__property System::UnicodeString ServerMainAddress
```

Use the ServerMainAddress property to specify the IP address that the database server should listen on for regular data connections when the EngineType property is set to etServer. A blank value (the default) indicates that the database server should listen on all available IP addresses from the operating system.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.ServerMainPort Property

```
__property int ServerMainPort
```

Use the ServerMainPort property to specify the port that the database server should listen on for regular data connections when the EngineType property is set to etServer. The default value is 12005.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.ServerMainThreadCacheSize Property

```
__property int ServerMainThreadCacheSize
```

Use the ServerMainThreadCacheSize property to specify the total number of threads that should be cached by the database server for regular data connections when the EngineType property is set to etServer. The default value is 10. Caching threads helps improve connection times by eliminating the need to constantly create and destroy threads as remote sessions connect to and disconnect from the database server.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.ServerName Property

```
__property System::UnicodeString ServerName
```

Use the ServerName property to specify the name of the database server when the EngineType property is set to etServer. The default value is "DBSRVR". This name is used when a remote session asks for it using the TDBISAMSession GetRemoteServerName method.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.SessionCount Property

```
__property int SessionCount
```

Use the SessionCount property to determine how many sessions are currently in use in DBISAM.

TDBISAMEngine.SessionList Property

```
__property TDBISAMSession* SessionList[const System::UnicodeString SessionName]
```

Use the SessionList property to access a given TDBISAMSession component by name. The name of a session is specified via the TDBISAMSession SessionName property.

Note
This property only applies when the EngineType property is set to etClient.

TDBISAMEngine.Sessions Property

```
__property TDBISAMSession* Sessions[int Index]
```

Use the Sessions property to access a given TDBISAMSession component by index. The Index parameter must be in the range of zero to the current value of the Count property minus one.

Note
This property only applies when the EngineType property is set to etClient.

TDBISAMEngine.StoreActive Property

```
__property bool StoreActive
```

Use the StoreActive property to determine if the DBISAM engine should store the current value of its Active property, and subsequently, the Active/Connected property values of all other DBISAM components such as the TDBISAMDatabase, TDBISAMTable, and TDBISAMQuery components, in the owner form or data module. The default value for this property is True.

Setting this property to False will ensure that you never run into the situation where the TDBISAMEngine component's Active property is automatically set to True (its design-time state) when the owning form/data module is created at runtime. This is a common problem when a developer is working with the DBISAM components at design-time, and then compiles the application with one or more of the DBISAM components' Active/Connected property set to True. The end result is usually many DBISAM runtime errors caused by the fact that the DBISAM engine has not been configured for the target machine and operating system, but rather is still configured for the developer's machine and operating system.

TDBISAMEngine.TableBlobBackupExtension Property

```
__property System::UnicodeString TableBlobBackupExtension
```

Use the TableBlobBackupExtension to specify the file extension used by the engine for backup copies of the physical BLOB file that makes up part of a logical DBISAM table. Backup files are created when altering the structure of a table or optimizing the table. The default value is ".bbk". Be sure to always include the filename extension separator (.) when specifying the file extension.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableBlobExtension Property

```
__property System::UnicodeString TableBlobExtension
```

Use the TableBlobExtension to specify the file extension used by the engine for the physical BLOB file that makes up part of a logical DBISAM table. The default value is ".blb". Be sure to always include the filename extension separator (.) when specifying the file extension.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableBlobTempExtension Property

```
__property System::UnicodeString TableBlobTempExtension
```

Use the TableIndexTempExtension to specify the file extension used by the engine for the physical BLOB file that makes up part of a logical DBISAM temporary table. Temporary tables are created when altering the structure of a table, optimizing the table, or for canned SQL result sets. The default value is ".blb". Be sure to always include the filename extension separator (.) when specifying the file extension.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableBlobUpgradeExtension Property

```
__property System::UnicodeString TableBlobUpgradeExtension
```

Use the TableBlobUpgradeExtension to specify the file extension used by the engine for backup copies of the physical BLOB file that makes up part of a logical DBISAM table. Backup files that use this file extensions are created specifically when upgrading a table from a prior version format to the latest table format. The default value is ".bup". Be sure to always include the filename extension separator (.) when specifying the file extension.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableDataBackupExtension Property

```
__property System::UnicodeString TableDataBackupExtension
```

Use the TableDataBackupExtension to specify the file extension used by the engine for backup copies of the physical data file that makes up part of a logical DBISAM table. Backup files are created when altering the structure of a table or optimizing the table. The default value is ".dbk". Be sure to always include the filename extension separator (.) when specifying the file extension.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableDataExtension Property

```
__property System::UnicodeString TableDataExtension
```

Use the TableDataExtension to specify the file extension used by the engine for the physical data file that makes up part of a logical DBISAM table. The default value is ".dat". Be sure to always include the filename extension separator (.) when specifying the file extension.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableDataTempExtension Property

```
__property System::UnicodeString TableDataTempExtension
```

Use the TableIndexTempExtension to specify the file extension used by the engine for the physical data file that makes up part of a logical DBISAM temporary table. Temporary tables are created when altering the structure of a table, optimizing the table, or for canned SQL result sets. The default value is ".dat". Be sure to always include the filename extension separator (.) when specifying the file extension.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableDataUpgradeExtension Property

```
__property System::UnicodeString TableDataUpgradeExtension
```

Use the TableDataUpgradeExtension to specify the file extension used by the engine for backup copies of the physical data file that makes up part of a logical DBISAM table. Backup files that use this file extensions are created specifically when upgrading a table from a prior version format to the latest table format. The default value is ".dup". Be sure to always include the filename extension separator (.) when specifying the file extension.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableFilterIndexThreshhold Property

```
__property System::Byte TableFilterIndexThreshhold
```

This property should only be modified when instructed to do so by Elevate Software in order to help tune a performance problem with filters or live queries.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableIndexBackupExtension Property

```
__property System::UnicodeString TableIndexBackupExtension
```

Use the TableIndexBackupExtension to specify the file extension used by the engine for backup copies of the physical index file that makes up part of a logical DBISAM table. Backup files are created when altering the structure of a table or optimizing the table. The default value is ".ibk". Be sure to always include the filename extension separator (.) when specifying the file extension.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableIndexExtension Property

```
__property System::UnicodeString TableIndexExtension
```

Use the TableIndexExtension to specify the file extension used by the engine for the physical index file that makes up part of a logical DBISAM table. The default value is ".idx". Be sure to always include the filename extension separator (.) when specifying the file extension.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableIndexTempExtension Property

```
__property System::UnicodeString TableIndexTempExtension
```

Use the TableIndexTempExtension to specify the file extension used by the engine for the physical index file that makes up part of a logical DBISAM temporary table. Temporary tables are created when altering the structure of a table, optimizing the table, or for canned SQL result sets. The default value is ".idx". Be sure to always include the filename extension separator (.) when specifying the file extension.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableIndexUpgradeExtension Property

```
__property System::UnicodeString TableIndexUpgradeExtension
```

Use the TableIndexUpgradeExtension to specify the file extension used by the engine for backup copies of the physical index file that makes up part of a logical DBISAM table. Backup files that use this file extensions are created specifically when upgrading a table from a prior version format to the latest table format. The default value is ".iup". Be sure to always include the filename extension separator (.) when specifying the file extension.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableMaxReadLockCount Property

```
__property System::Word TableMaxReadLockCount
```

Use the TableMaxReadLockCount property to control the maximum number of read locks that DBISAM can acquire during a table scan to satisfy an un-optimized filter or query condition. The default value is 100 locks, and the maximum value is 65535. Increasing this property will increase concurrency, but at the cost of a decrease in performance. Decreasing this property will increase performance, but at the cost of a decrease in concurrency.

TDBISAMEngine.TableReadLockTimeout Property

```
__property System::Word TableReadLockTimeout
```

This property should only be modified when instructed to do so by Elevate Software in order to help with locking issues in a particular environment.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableTransLockTimeout Property

```
__property System::Word TableTransLockTimeout
```

This property should only be modified when instructed to do so by Elevate Software in order to help with locking issues in a particular environment.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.TableWriteLockTimeout Property

```
__property System::Word TableWriteLockTimeout
```

This property should only be modified when instructed to do so by Elevate Software in order to help with locking issues in a particular environment.

Note
This property cannot be set while the engine is active and the TDBISAMEngine Active property is True.

TDBISAMEngine.AddServerDatabase Method

```
void __fastcall AddServerDatabase(const System::UnicodeString
      DatabaseName, const System::UnicodeString DatabaseDescription,
      const System::UnicodeString ServerPath)
```

Call the AddServerDatabase method to add a new database to a database server. Use the DatabaseName parameter to specify the new database name, the DatabaseDescription parameter to give it a description, and the ServerPath parameter to specify the physical path to the tables, relative to the database server.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.AddServerDatabaseUser Method

```
void __fastcall AddServerDatabaseUser(const
      System::UnicodeString DatabaseName, const System::UnicodeString
      AuthorizedUser, TDatabaseRights RightsToAssign)
```

Call the AddServerDatabaseUser method to add rights for an existing user to an existing database on a database server. Use the DatabaseName parameter to specify the existing database name, the AuthorizedUser parameter to specify the existing user, and the RightsToAssign parameter to specify the rights to give to the user for the database. You may use a wildcard (*) for the AuthorizedUser parameter. For example, you could specify just "*" for all users or "Accounting*" for all users whose user name begins with "Accounting".

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.AddServerEvent Method

```
void __fastcall AddServerEvent(const System::UnicodeString
      EventName, const System::UnicodeString EventDescription,
      TEventRunType EventRunType, System::TDateTime EventStartDate,
      System::TDateTime EventEndDate, System::TDateTime EventStartTime,
       System::TDateTime EventEndTime, System::Word EventInterval,
      TEventDays EventDays, TEventDayOfMonth EventDayOfMonth,
      TEventDayOfWeek EventDayOfWeek, const TEventMonths &EventMonths)
```

Call the AddServerEvent method to add a new scheduled event to a database server. Use the EventName parameter to specify the new event name, the EventDescription parameter to give it a description, the EventRunType parameter to specify how the event should be run, the EventStartDate and EventEndDate parameter to specify the dates during which the event should be run, the EventStartTime and EventEndTime parameters to specify the time of day during which the event can be run, the EventInterval to specify how often the event should be run (actual interval unit depends upon the EventRunType, and the EventDays, EventDayOfMonth, EventDayOfWeek, and EventMonths parameters to specify on what day of the week or month the event should be run. The following describes which parameters are required for each possible EventRunType value (all run types require the EventStartDate, EventEndDate, EventStartTime, and EventEndTime parameters):

Run Type	Parameters Needed
rtOnce	No Other Parameters
rtHourly	EventInterval (Hours)
rtDaily	EventInterval (Days)
rtWeekly	EventInterval (Weeks) EventDays **Note** The EventDays parameter specifies which days of the week to run on, with day 1 being Sunday and day 7 being Saturday. Just set the array index of the desired day to True to cause the event to run on that day.
rtMonthly	EventDayOfMonth EventDayOfWeek EventMonths

	Note The EventDayOfMonth parameter specifies which day of the month to run on, a numbered day (1-31) or a specific day (Sunday-Saturday) of the 1st, 2nd, 3rd, or 4th week specified by the EventDayOfWeekParameter. The EventMonths parameter specifies which months of the year to run on, with month 1 being January and month 12 being December. Just set the array index of the desired month to True to cause the event to run on that month.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.AddServerProcedure Method

```
void __fastcall AddServerProcedure(const System::UnicodeString
      ProcedureName, const System::UnicodeString ProcedureDescription)
```

Call the AddServerProcedure method to add a new server-side procedure to a database server. Use the ProcedureName parameter to specify the new procedure name and the ProcedureDescription parameter to give it a description. This method only identifies the procedure to the database server for the purposes of allowing user rights to be assigned to the server-side procedure. The actual server-side procedure itself must be implemented via a TDBISAMEngine OnServerProcedure event handler.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.AddServerProcedureUser Method

```
void __fastcall AddServerProcedureUser(const
      System::UnicodeString ProcedureName, const System::UnicodeString
      AuthorizedUser, TProcedureRights RightsToAssign)
```

Call the AddServerProcedureUser method to add rights for an existing user to an existing server-side procedure on a database server. Use the ProcedureName parameter to specify the existing server-side procedure name, the AuthorizedUser parameter to specify the existing user, and the RightsToAssign parameter to specify the rights to give to the user for the procedure. You may use a wildcard (*) for the AuthorizedUser parameter. For example, you could specify just "*" for all users or "Accounting*" for all users whose user name begins with "Accounting".

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.AddServerUser Method

```
void __fastcall AddServerUser(const System::UnicodeString
      UserName, const System::UnicodeString UserPassword, const
      System::UnicodeString UserDescription, bool IsAdministrator =
      false, System::Word MaxConnections = (System::Word)(0x64))
```

Call the AddServerUser method to add a new user to a database server. Use the UserName parameter to specify the new user name, the UserPassword parameter to specify the user's password, the UserDescription parameter to specify a description of the user, the IsAdministrator parameter to indicate whether the new user is an administrator, and the MaxConnections property is used to specify how many concurrent connections this user is allowed to have at any given time.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.AnsiStrToBoolean Method

```
bool __fastcall AnsiStrToBoolean(const System::UnicodeString
      Value)
```

Use the AnsiStrToBoolean method to convert a string that contains an ANSI-formatted boolean value to an actual Boolean value. All SQL and filter expressions in DBISAM require ANSI-formatted boolean values, which are TRUE and FALSE (case-insensitive).

TDBISAMEngine.AnsiStrToCurr Method

```
System::Currency __fastcall AnsiStrToCurr(const
      System::UnicodeString Value)
```

Use the AnsiStrToCurr method to convert a string that contains an ANSI-formatted BCD value to an actual Currency value. All SQL and filter expressions in DBISAM require ANSI-formatted BCD values, which use the period (.) as the decimal separator.

TDBISAMEngine.AnsiStrToDate Method

```
System::TDateTime __fastcall AnsiStrToDate(const
      System::UnicodeString Value)
```

Use the AnsiStrToDate method to convert a string that contains an ANSI-formatted date value to an actual TDateTime value. All SQL and filter expressions in DBISAM require ANSI-formatted date values, which use the 'yyyy-mm-dd' format.

TDBISAMEngine.AnsiStrToDateTime Method

```
System::TDateTime __fastcall AnsiStrToDateTime(const
      System::UnicodeString Value)
```

Use the AnsiStrToDateTime method to convert a string that contains an ANSI-formatted timestamp value to an actual TDateTime value. All SQL and filter expressions in DBISAM require ANSI-formatted timestamp values, which use the 'yyyy-mm-dd hh:mm:ss.zzz am/pm' format.

TDBISAMEngine.AnsiStrToFloat Method

```
System::Extended __fastcall AnsiStrToFloat(const
      System::UnicodeString Value)
```

Use the AnsiStrToFloat method to convert a string that contains an ANSI-formatted float value to an actual Double value. All SQL and filter expressions in DBISAM require ANSI-formatted float values, which use the period (.) as the decimal separator.

TDBISAMEngine.AnsiStrToTime Method

```
System::TDateTime __fastcall AnsiStrToTime(const
      System::UnicodeString Value)
```

Use the AnsiStrToTime method to convert a string that contains an ANSI-formatted time value to an actual TDateTime value. All SQL and filter expressions in DBISAM require ANSI-formatted time values, which use the 'hh:mm:ss.zzz am/pm' format.

TDBISAMEngine.BooleanToAnsiStr Method

```
System::UnicodeString __fastcall BooleanToAnsiStr(bool Value)
```

Use the BooleanToAnsiStr method to convert a Boolean value to an ANSI-formatted boolean value string. All SQL and filter expressions in DBISAM require ANSI-formatted boolean values, which are TRUE and FALSE (case-insensitive).

TDBISAMEngine.BuildWordList Method

```
void __fastcall BuildWordList(const System::UnicodeString
      TableName, const System::UnicodeString FieldName,
      System::UnicodeString WordBuffer, TDBISAMStringList* WordList,
      const System::UnicodeString SpaceChars, const
      System::UnicodeString IncludeChars, bool Occurrences, bool
      PartialWords)
```

Use the BuildWordList method to populate the WordList parameter with the word tokens that are parsed from the WordBuffer string parameter passed to the method, using the SpaceChars and IncludeChars parameters to control the parsing. The TableName and FieldName parameters specify which table and field the WordBuffer string applies to. The reason for this information is that it is passed on to the TDBISAMEngine OnTextIndexFilter and OnTextIndexTokenFilter events in order to give the event handlers attached to these events the proper information about how to perform any custom filtering. The Occurrences parameter is used to specify whether you wish to have the Objects property for each string list item (search word) populated with the number of occurrences of that word or with the position of the start of that word (0-based) in the WordBuffer parameter. You'll have to cast these integer values from TObject values when accessing them via the Objects property. The PartialWords parameter is used to specify whether partial words should be allowed and asterisks in words should be left intact even if they are designated as a space character or not designated as an include character. This parameter is useful when using the BuildWordList method for parsing a search string that includes wildcards.

Note
The WordList parameter is a special descendant of the TStringList object called TDBISAMStringList for performing locale-specific sorting and searching of a string list. The TDBIAMStringList object contains an additional LocaleID property that can be set to match the locale of the table being searched. The WordList parameter is where the results of this procedure are stored, and this parameter is cleared during the operation of the procedure.

TDBISAMEngine.ConvertIDToLocaleConstant Method

```
System::UnicodeString __fastcall ConvertIDToLocaleConstant(int
      LocaleID)
```

Use the ConvertIDToLocaleConstant method to convert a locale ID, which is a 32-bit signed integer, to its equivalent string name. Many SQL statements in DBISAM require the name of a locale as part of a LOCALE clause, and this method allows one to easily generate dynamic SQL statements that require a LOCALE clause.

TDBISAMEngine.ConvertLocaleConstantToID Method

```
int __fastcall ConvertLocaleConstantToID(const
      System::UnicodeString LocaleConstant)
```

Use the ConvertLocaleConstantToConstant method to convert a locale name to its equivalent ID, which is a 32-bit signed integer. Many SQL statements in DBISAM require the name of a locale as part of a LOCALE clause, and this method can help convert a locale name back to its proper ID if one is parsing SQL statements for application-specific purposes.

TDBISAMEngine.CurrToAnsiStr Method

```
System::UnicodeString __fastcall CurrToAnsiStr(System::Currency
      Value)
```

Use the CurrToAnsiStr method to convert a Currency value to an ANSI-formatted BCD value string. All SQL and filter expressions in DBISAM require ANSI-formatted BCD values, which use the period (.) as the decimal separator.

TDBISAMEngine.DateTimeToAnsiStr Method

```
System::UnicodeString __fastcall
      DateTimeToAnsiStr(System::TDateTime Value, bool MilitaryTime)
```

Use the DateTimeToAnsiStr method to convert a TDateTime value to an ANSI-formatted timestamp value string. All SQL and filter expressions in DBISAM require ANSI-formatted timestamp values, which use the 'yyyy-mm-dd hh:mm:ss.zzz am/pm' format.

TDBISAMEngine.DateToAnsiStr Method

```
System::UnicodeString __fastcall DateToAnsiStr(System::TDateTime
      Value)
```

Use the DateToAnsiStr method to convert a TDateTime value to an ANSI-formatted date value string. All SQL and filter expressions in DBISAM require ANSI-formatted date values, which use the 'yyyy-mm-dd' format.

TDBISAMEngine.DeleteServerDatabase Method

```
void __fastcall DeleteServerDatabase(const System::UnicodeString
      DatabaseName)
```

Call the DeleteServerDatabase method to remove an existing database from a database server. Use the DatabaseName parameter to specify the existing database name.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.DeleteServerDatabaseUser Method

```
void __fastcall DeleteServerDatabaseUser(const
      System::UnicodeString DatabaseName, const System::UnicodeString
      AuthorizedUser)
```

Call the DeleteServerDatabaseUser method to remove rights for an existing user to an existing database on a database server. Use the DatabaseName parameter to specify the existing database name and the AuthorizedUser parameter to specify the existing user.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.DeleteServerEvent Method

```
void __fastcall DeleteServerEvent(const System::UnicodeString
      EventName)
```

Call the DeleteServerEvent method to remove an existing scheduled event from a database server. Use the EventName parameter to specify the existing scheduled event name.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.DeleteServerProcedure Method

```
void __fastcall DeleteServerProcedure(const
      System::UnicodeString ProcedureName)
```

Call the DeleteServerProcedure method to remove an existing server-side procedure from a database server. Use the ProcedureName parameter to specify the existing server-side procedure name.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.DeleteServerProcedureUser Method

```
void __fastcall DeleteServerProcedureUser(const
      System::UnicodeString ProcedureName, const System::UnicodeString
      AuthorizedUser)
```

Call the DeleteServerProcedureUser method to remove rights for an existing user to an existing server-side procedure on a database server. Use the ProcedureName parameter to specify the existing server-side procedure name and the AuthorizedUser parameter to specify the existing user.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.DeleteServerUser Method

```
void __fastcall DeleteServerUser(const System::UnicodeString
      UserName)
```

Call the DeleteServerUser method to remove an existing user from a database server. Use the UserName parameter to specify the existing user name.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.DisconnectServerSession Method

```
bool __fastcall DisconnectServerSession(int SessionID)
```

Call the DisconnectServerSession method to disconnect a specific session on a database server. Disconnecting a session only terminates its connection, it does not remove the session completely from the database server nor does it release any resources for the session other than the thread used for the connection and the connection itself at the operating system level. Use the SessionID parameter to specify the session ID to disconnect. You can get the session ID for a particular session by using the GetServerSessionCount and the GetServerSessionInfo methods.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.FindSession Method

```
TDBISAMSession* __fastcall FindSession(const
      System::UnicodeString SessionName)
```

Use the FindSession method to search the list of TDBISAMSession components for a specified session name. SessionName specifies the session to search for.

FindSession compares the SessionName parameter to the SessionName property for each TDBISAMSession component in the available list of sessions in the engine. If a match is found, FindSession returns a reference to the applicable TDBISAMSession component. If an application passes an empty string in the SessionName parameter, FindSession returns the default global TDBISAMSession, Session. If a match is not found, FindSession returns nil.

TDBISAMEngine.FloatToAnsiStr Method

```
System::UnicodeString __fastcall FloatToAnsiStr(System::Extended
      Value)
```

Use the FloatToAnsiStr method to convert a Double value to an ANSI-formatted float value string. All SQL and filter expressions in DBISAM require ANSI-formatted float values, which use the period (.) as the decimal separator.

TDBISAMEngine.GetDefaultTextIndexParams Method

```
void __fastcall
      GetDefaultTextIndexParams(System::Classes::TStrings*
      StopWordsList, System::UnicodeString &TextSpaceChars,
      System::UnicodeString &TextIncludeChars)
```

Use the GetDefaultTextIndexParams to retrieve the default full text indexing parameters. When this method is done, the StopWordsList parameter will contains a list of the default stop words, the TextSpaceChars parameter will contain all of the default space characters, and the TextIncludeChars parameter will contains all of the default include characters.

TDBISAMEngine.GetLocaleNames Method

```
void __fastcall GetLocaleNames(System::Classes::TStrings* List)
```

Use the GetLocaleNames method to populate a list of all locale names and their IDs that are available through the operating system. Use the Objects property of the string list parameter to access the 32-bit integer ID of a given locale. However, you will have to cast it from a TObject to an integer first. Many SQL statements in DBISAM require the name of a locale as part of a LOCALE clause, and this method can allow the application to offer the user a choice of the desired locale.

Note
This method will only return one name, 'ANSI Standard', under Kylix and Linux since that is the only locale available currently under Linux for DBISAM.

TDBISAMEngine.GetServerConfig Method

```
void __fastcall GetServerConfig(bool &DenyLogins, System::Word
      &MaxConnections, System::Word &ConnectTimeout, System::Word
      &DeadSessionInterval, System::Word &DeadSessionExpires,
      System::Word &MaxDeadSessions, System::UnicodeString
      &TempDirectory, System::Classes::TStrings* AuthorizedAddresses,
      System::Classes::TStrings* BlockedAddresses)
```

Call the GetServerConfig method to retrieve the current configuration settings for a database server. Please see the ModifyServerConfig method for more information on the parameters returned from this method.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer. Also, if the maximum number of connections returned via this method is lower than what was attempted to be configured via the ModifyServerConfig method, the ServerLicensedConnections property has caused it to be lowered.

TDBISAMEngine.GetServerConnectedSessionCount Method

```
int __fastcall GetServerConnectedSessionCount(void)
```

Call the GetServerConnectedSessionCount method to retrieve the total number of connected sessions on a database server. Sessions that are present on the server, but not connected, are not reported in this figure. To get a total count of the number of sessions on the database server use the GetServerSessionCount method instead.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerDatabase Method

```
void __fastcall GetServerDatabase(const System::UnicodeString
      DatabaseName, System::UnicodeString &DatabaseDescription,
      System::UnicodeString &ServerPath)
```

Call the GetServerDatabase method to retrieve information about an existing database from a database server. Use the DatabaseName parameter to specify the existing database name.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerDatabaseNames Method

```
void __fastcall
      GetServerDatabaseNames(System::Classes::TStrings* List)
```

Call the GetServerDatabaseNames method to retrieve a list of databases defined on a database server.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerDatabaseUser Method

```
void __fastcall GetServerDatabaseUser(const
      System::UnicodeString DatabaseName, const System::UnicodeString
      AuthorizedUser, TDatabaseRights &UserRights)
```

Call the GetServerDatabaseUser method to retrieve the rights for an existing user to an existing database on a database server. Use the DatabaseName parameter to specify the existing database name and the AuthorizedUser parameter to specify the existing user.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerDatabaseUserNames Method

```
void __fastcall GetServerDatabaseUserNames(const
      System::UnicodeString DatabaseName, System::Classes::TStrings*
      List)
```

Call the GetServerDatabaseUserNames method to retrieve a list of existing users defined with rights for an existing database on a database server. Use the DatabaseName parameter to specify an existing database from which to retrieve a list of users.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerEvent Method

```
void __fastcall GetServerEvent(const System::UnicodeString
      EventName, System::UnicodeString &EventDescription,
      TEventRunType &EventRunType, System::TDateTime &EventStartDate,
      System::TDateTime &EventEndDate, System::TDateTime
      &EventStartTime, System::TDateTime &EventEndTime, System::Word
      &EventInterval, TEventDays &EventDays, TEventDayOfMonth
      &EventDayOfMonth, TEventDayOfWeek &EventDayOfWeek, TEventMonths
      &EventMonths, System::TDateTime &EventLastRun)
```

Call the GetServerEvent method to retrieve information about an existing scheduled event from a database server. Use the EventName parameter to specify the existing event name. Please see the AddServerEvent method for more information on the parameters returned from this method. The EventLastRun parameter specifies the last date and time that the event was successfully run.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerEventNames Method

```
void __fastcall GetServerEventNames(System::Classes::TStrings*
      List)
```

Call the GetServerEventNames method to retrieve a list of scheduled events defined on a database server.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerLogCount Method

```
int __fastcall GetServerLogCount(void)
```

Call the GetServerLogCount method to retrieve the total count of log records available in the current log on a database server.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerLogRecord Method

```
TLogRecord __fastcall GetServerLogRecord(int Number)
```

Call the GetServerLogRecord method to retrieve the Nth log record from the current log on a database server.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerMemoryUsage Method

```
double __fastcall GetServerMemoryUsage(void)
```

Call the GetServerMemoryUsage method to retrieve the total amount of memory (in megabytes) currently allocated by a database server.

Note
This method has been deprecated and always returns 0 as of version 4.17 of DBISAM and the introduction of the new memory manager used in the DBISAM database server. Please see the Replacement Memory Manager topic for more information.

TDBISAMEngine.GetServerProcedure Method

```
void __fastcall GetServerProcedure(const System::UnicodeString
      ProcedureName, System::UnicodeString &ProcedureDescription)
```

Call the GetServerProcedure method to retrieve information about an existing server-side procedure from a database server. Use the ProcedureName parameter to specify the existing server-side procedure name.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerProcedureNames Method

```
void __fastcall
      GetServerProcedureNames(System::Classes::TStrings* List)
```

Call the GetServerProcedureNames method to retrieve a list of server-side procedures defined on a database server.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerProcedureUser Method

```
void __fastcall GetServerProcedureUser(const
      System::UnicodeString ProcedureName, const System::UnicodeString
      AuthorizedUser, TProcedureRights &UserRights)
```

Call the GetServerProcedureUser method to retrieve the rights for an existing user to an existing server-side procedure on a database server. Use the ProcedureName parameter to specify the existing server-side procedure name and the AuthorizedUser parameter to specify the existing user.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerProcedureUserNames Method

```
void __fastcall GetServerProcedureUserNames(const
      System::UnicodeString ProcedureName, System::Classes::TStrings*
      List)
```

Call the GetServerProcedureUserNames method to retrieve a list of existing users defined with rights for an existing server-side procedure on a database server. Use the ProcedureName parameter to specify an existing server-side procedure from which to retrieve a list of users.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerSessionCount Method

```
int __fastcall GetServerSessionCount(void)
```

Call the GetServerSessionCount method to retrieve the total number of sessions on a database server. To get a total count of just the number of connected sessions on a database server use the GetServerConnectedSessionCount method instead.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerSessionInfo Method

```
bool __fastcall GetServerSessionInfo(int SessionNum, int
      &SessionID, System::TDateTime &CreatedOn, System::TDateTime
      &LastConnectedOn, System::UnicodeString &UserName,
      System::UnicodeString &UserAddress, bool &Encrypted,
      System::UnicodeString &LastUserAddress)
```

Call the GetServerSessionInfo method to retrieve session information for a specific session on a database server. The SessionNum parameter indicates the session number for which to retrieve the session information. This number represents the logical position of a given session in the list of sessions on a database server, from 1 to the return value of the GetServerSessionCount method. The SessionID parameter returns unique ID assigned to the session by the database server. The CreatedOn parameter returns the date and time when the session was created on the database server. The LastConnectedOn parameter returns the date and time when the session was last connected to the database server. The UserName parameter returns the name of the user that created the session on the database server. The UserAddress parameter returns the IP address of the user that created the session on the database server. If the session is not currently connected, then this parameter will be blank. The Encrypted parameter returns whether the session is encrypted or not. The LastUserAddress parameter returns the last known IP address of the session, regardless of whether the session is connected or not. This parameter is useful for determining the location of a workstation that still has an active session but has disconnected.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerUpTime Method

```
__int64 __fastcall GetServerUpTime(void)
```

Call the GetServerUpTime method to retrieve the number of seconds that the database server has been active and accepting new connections.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerUser Method

```
void __fastcall GetServerUser(const System::UnicodeString
      UserName, System::UnicodeString &UserPassword,
      System::UnicodeString &UserDescription, bool &IsAdministrator,
      System::Word &MaxConnections)
```

Call the GetServerUser method to retrieve information about an existing user from a database server. Use the UserName parameter to specify the existing user name.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerUserNames Method

```
void __fastcall GetServerUserNames(System::Classes::TStrings*
      List)
```

Call the GetServerUserNames method to retrieve a list of users defined on a database server.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetServerUTCDateTime Method

```
System::TDateTime __fastcall GetServerUTCDateTime(void)
```

Call the GetServerUTCDateTime method to retrieve the universal coordinate date and time from a database server. This is especially useful if you are accessing a database server in a different time zone and wish to get the date and time in a standard format that doesn't need to take into account the local server time offset.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.GetSessionNames Method

```
void __fastcall GetSessionNames(System::Classes::TStrings* List)
```

Call the GetSessionNames method to populate a string list with the names of all available TDBISAMSession components. The List parameter is a string list object, created and maintained by the application, into which to store session names. The names returned by GetSessionNames correspond to the SessionName properties of all available TDBISAMSession components.

TDBISAMEngine.IsValidLocale Method

```
bool __fastcall IsValidLocale(int LocaleID)
```

Use the IsValidLocale method to verify whether a specific locale ID, which is a 32-bit signed integer, is valid and available through the operating system.

Note
This method will only consider one locale, 'ANSI Standard' (ID of 0), as valid under Kylix and Linux since that is the only locale available currently under Linux for DBISAM.

TDBISAMEngine.IsValidLocaleConstant Method

```
bool __fastcall IsValidLocaleConstant(const
      System::UnicodeString LocaleConstant)
```

Use the IsValidLocaleConstant method to verify whether a specific locale name is valid and available through the operating system. Many SQL statements in DBISAM require the name of a locale as part of a LOCALE clause, and this method can help verify a locale name prior to being used in a dynamic SQL statement.

Note
This method will only consider one locale, 'ANSI Standard' (ID of 0), as valid under Kylix and Linux since that is the only locale available currently under Linux for DBISAM.

TDBISAMEngine.ModifyServerConfig Method

```
void __fastcall ModifyServerConfig(bool DenyLogins, System::Word
      MaxConnections, System::Word ConnectTimeout, System::Word
      DeadSessionInterval, System::Word DeadSessionExpires,
      System::Word MaxDeadSessions, const System::UnicodeString
      TempDirectory, System::Classes::TStrings* AuthorizedAddresses,
      System::Classes::TStrings* BlockedAddresses)
```

Call the ModifyServerConfig method to modify the current configuration settings for a database server. The DenyLogins parameter indicates whether any new logins are denied on the database server. The MaxConnections parameter indicates the maximum allowable number of connected sessions (not total sessions) on the database server. The ConnectTimeout parameter indicates how long a session is allowed to remain idle before the session is disconnected automatically by the database server. The DeadSessionInterval parameter indicates how often the database server should check for dead sessions (sessions that have been disconnected for DeadSessionExpires seconds). The DeadSessionExpires parameter indicates when a disconnected session is considered "dead" based upon the number of seconds since it was last connected. Specifying 0 for this parameter will cause the database server to never consider disconnected sessions as dead and instead will keep them around based upon the MaxDeadSessions parameter alone. The MaxDeadSessions parameter indicates how many dead sessions are allowed on the database server before the database server will start removing dead sessions in oldest-first order. The TempDirectory parameter indicates where temporary tables are stored relative to the database server. This setting is global for all users. The AuthorizedAddresses and BlockedAddresses parameters are lists of IP addresses that specify which IP addresses are allowed or blocked from accessing the database server. Both of these accept the use of a leading * wildcard when specifying IP addresses.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.ModifyServerDatabase Method

```
void __fastcall ModifyServerDatabase(const System::UnicodeString
      DatabaseName, const System::UnicodeString DatabaseDescription,
      const System::UnicodeString ServerPath)
```

Call the ModifyServerDatabase method to modify information about an existing database on a database server. Use the DatabaseName parameter to specify the existing database.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.ModifyServerDatabaseUser Method

```
void __fastcall ModifyServerDatabaseUser(const
      System::UnicodeString DatabaseName, const System::UnicodeString
      AuthorizedUser, TDatabaseRights RightsToAssign)
```

Call the ModifyServerDatabaseUser method to modify the rights for an existing user to an existing database on a database server. Use the DatabaseName parameter to specify the existing database name and the AuthorizedUser parameter to specify the existing user. You may use a wildcard (*) for the AuthorizedUser parameter , such as specifying just '*' for all users or 'Accounting*' for all users whose user name begins with 'Accounting'.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.ModifyServerEvent Method

```
void __fastcall ModifyServerEvent(const System::UnicodeString
      EventName, const System::UnicodeString EventDescription,
      TEventRunType EventRunType, System::TDateTime EventStartDate,
      System::TDateTime EventEndDate, System::TDateTime EventStartTime,
       System::TDateTime EventEndTime, System::Word EventInterval,
      TEventDays EventDays, TEventDayOfMonth EventDayOfMonth,
      TEventDayOfWeek EventDayOfWeek, const TEventMonths &EventMonths)
```

Call the ModifyServerEvent method to modify information about an existing scheduled event on a database server. Use the EventName parameter to specify the existing event name. Please see the AddServerEvent method for more information on the parameters returned from this method.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.ModifyServerProcedure Method

```
void __fastcall ModifyServerProcedure(const
      System::UnicodeString ProcedureName, const System::UnicodeString
      ProcedureDescription)
```

Call the ModifyServerProcedure method to modify information about an existing server-side procedure on a database server. Use the ProcedureName parameter to specify the existing server-side procedure.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.ModifyServerProcedureUser Method

```
void __fastcall ModifyServerProcedureUser(const
      System::UnicodeString ProcedureName, const System::UnicodeString
      AuthorizedUser, TProcedureRights RightsToAssign)
```

Call the ModifyServerProcedureUser method to modify the rights for an existing user to an existing server-side procedure on a database server. Use the ProcedureName parameter to specify the existing server-side procedure name and the AuthorizedUser parameter to specify the existing user. You may use a wildcard (*) for the AuthorizedUser parameter , such as specifying just '*' for all users or 'Accounting*' for all users whose user name begins with 'Accounting'.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.ModifyServerUser Method

```
void __fastcall ModifyServerUser(const System::UnicodeString
      UserName, const System::UnicodeString UserPassword, const
      System::UnicodeString UserDescription, bool IsAdministrator =
      false, System::Word MaxConnections = (System::Word)(0x64))
```

Call the ModifyServerUser method to modify information about an existing user on a database server. Use the UserName parameter to specify the existing user.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.ModifyServerUserPassword Method

```
void __fastcall ModifyServerUserPassword(const
      System::UnicodeString UserName, const System::UnicodeString
      UserPassword)
```

Call the ModifyServerUserPassword method to modify the password for the current user logged in to the database server. Use the UserName parameter to specify the current user name. This method is only valid for changing the password for the current user.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.OpenSession Method

```
TDBISAMSession* __fastcall OpenSession(const
      System::UnicodeString SessionName)
```

Call the OpenSession method to make an existing TDBISAMSession component active, or to create a new TDBISAMSession component and make it active. SessionName specifies the name of the session to open.

OpenSession calls the TDBISAMEngine FindSession method to see if the TDBISAMSession component specified in the SessionName parameter already exists. If it finds a match via the SessionName property of an existing TDBISAMSession component, it starts that session if necessary, and makes the session active. If OpenSession does not find an existing TDBISAMSession component with that name, it creates a new TDBISAMSession component using the name specified in the SessionName parameter, starts the session, and makes it active.

In either case, OpenSession returns the TDBISAMSession component.

TDBISAMEngine.QuotedSQLStr Method

```
System::UnicodeString __fastcall QuotedSQLStr(const
      System::UnicodeString Value)
```

Call the QuotedSQLStr method to format a string constant so that it can properly used as a literal constant in an SQL statement. This method converts escapes all single quotes and converts all characters less than #32 (space) into the #<ASCII value> syntax.

TDBISAMEngine.RemoveServerSession Method

```
bool __fastcall RemoveServerSession(int SessionID)
```

Call the RemoveServerSession method to completely remove a specific session on a database server. Removing a session not only terminates its connection, but it also removes the session completely and releases any resources for the session including the thread used for the connection and the connection itself at the operating system level. Use the SessionID parameter to specify the session ID to disconnect. You can get the session ID for a particular session by using the GetServerSessionCount and the GetServerSessionInfo methods.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer.

TDBISAMEngine.StartAdminServer Method

```
void __fastcall StartAdminServer(void)
```

Call the StartAdminServer method to cause the database server to start accepting administrative connections from remote sessions. You may call the StartAdminServer or StopAdminServer methods without removing existing administrative sessions. When stopping the administrative server, however, all administrative sessions will be automatically disconnected.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer. Also, this method is implicitly called when the Active is set to True.

TDBISAMEngine.StartMainServer Method

```
void __fastcall StartMainServer(void)
```

Call the StartMainServer method to cause the database server to start accepting regular data connections from remote sessions. You may call the StartMainServer or StopMainServer methods without removing existing sessions. When stopping the server, however, all sessions will be automatically disconnected.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer. Also, this method is implicitly called when the Active is set to True.

TDBISAMEngine.StopAdminServer Method

```
void __fastcall StopAdminServer(void)
```

Call the StopAdminServer method to cause the database server to stop accepting administrative connections from remote sessions. You may call the StartAdminServer or StopAdminServer methods without removing existing administrative sessions. When stopping the administrative server, however, all administrative sessions will be automatically disconnected.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer. Also, this method is implicitly called when the Active is set to False.

TDBISAMEngine.StopMainServer Method

```
void __fastcall StopMainServer(void)
```

Call the StopMainServer method to cause the database server to stop accepting regular data connections from remote sessions. You may call the StartMainServer or StopMainServer methods without removing existing sessions. When stopping the server, however, all sessions will be automatically disconnected.

Note
This method is only valid when the engine is running as a database server and the EngineType is set to etServer. Also, this method is implicitly called when the Active is set to False.

TDBISAMEngine.TDBISAMEngine Method

```
__fastcall virtual TDBISAMEngine(System::Classes::TComponent*
      AOwner)
```

You'll never need to call the TDBISAMEngine constructor since a global Engine function is always available in DBISAM.

TDBISAMEngine.TimeToAnsiStr Method

```
System::UnicodeString __fastcall TimeToAnsiStr(System::TDateTime
      Value, bool MilitaryTime)
```

Use the TimeToAnsiStr method to convert a TDateTime value to an ANSI-formatted time value string. All SQL and filter expressions in DBISAM require ANSI-formatted time values, which use the 'hh:mm:ss.zzz am/pm' format.

TDBISAMEngine.AfterDeleteTrigger Event

```
__property TTriggerEvent AfterDeleteTrigger
```

The AfterDeleteTrigger event is fired after the engine has deleted a record in a table. Assigning an event handler to this event allows you to perform processing after the deletion of any record in any table.

Note
You cannot modify the contents of the CurrentRecord parameter in this event. Also, this event is not triggered during any system processing such as Creating and Altering Tables or Optimizing Tables.

TDBISAMEngine.AfterInsertTrigger Event

```
__property TTriggerEvent AfterInsertTrigger
```

The AfterInsertTrigger event is fired after the engine has inserted a record in a table. Assigning an event handler to this event allows you to perform processing after the insertion of any record in any table.

Note
You cannot modify the contents of the CurrentRecord parameter in this event. Also, this event is not triggered during any system processing such as Creating and Altering Tables or Optimizing Tables.

TDBISAMEngine.AfterUpdateTrigger Event

```
__property TTriggerEvent AfterUpdateTrigger
```

The AfterUpdateTrigger event is fired after the engine has updated a record in a table. Assigning an event handler to this event allows you to perform processing after the update of any record in any table.

Note
You cannot modify the contents of the CurrentRecord parameter in this event. Also, this event is not triggered during any system processing such as Creating and Altering Tables or Optimizing Tables.

TDBISAMEngine.BeforeDeleteTrigger Event

```
__property TTriggerEvent BeforeDeleteTrigger
```

The BeforeDeleteTrigger event is fired before the engine deletes a record in a table. Assigning an event handler to this event allows you to perform processing before the deletion of any record in any table.

Note
You cannot modify the contents of the CurrentRecord parameter in this event. Also, this event is not triggered during any system processing such as Creating and Altering Tables or Optimizing Tables.

TDBISAMEngine.BeforeInsertTrigger Event

```
__property TTriggerEvent BeforeInsertTrigger
```

The BeforeInsertTrigger event is fired before the engine inserts a record in a table. Assigning an event handler to this event allows you to perform processing before the insertion of any record in any table, including modifying the contents of any field in the CurrentRecord parameter.

Note
This event is not triggered during any system processing such as Creating and Altering Tables or Optimizing Tables.

TDBISAMEngine.BeforeUpdateTrigger Event

```
__property TTriggerEvent BeforeUpdateTrigger
```

The BeforeUpdateTrigger event is fired before the engine updates a record in a table. Assigning an event handler to this event allows you to perform processing before the update of any record in any table, including modifying the contents of any field in the CurrentRecord parameter.

Note
This event is not triggered during any system processing such as Creating and Altering Tables or Optimizing Tables.

TDBISAMEngine.CommitTrigger Event

```
__property TEndTransactionTriggerEvent CommitTrigger
```

The CommitTrigger event is fired after the engine has committed a transaction for a database. Assigning an event handler to this event allows you to perform processing after the transaction commit.

TDBISAMEngine.OnCompress Event

```
__property TCompressEvent OnCompress
```

The OnCompress event is fired when the engine needs to compress a buffer. Please see the Compression topic for more information on the default compression in DBISAM.

Note
If you assign an event handler to this event to override the default compression, you should make sure that you also assign an event handler to the OnDecompress event. Failure to do so can cause serious problems such as access violations.

TDBISAMEngine.OnCryptoInit Event

```
__property TCryptoInitEvent OnCryptoInit
```

The OnCryptoInit event is fired when the engine needs to initialize the block ciper data for a specified key. Please see the Encryption topic for more information on the default block cipher encryption in DBISAM.

Note
If you assign an event handler to this event to override the default encryption initialization, you should make sure that you also assign an event handler to the OnCryptoReset, OnEncryptBlock, OnDecryptBlock events. Failure to do so can cause serious problems such as access violations.

TDBISAMEngine.OnCryptoReset Event

```
__property TCryptoResetEvent OnCryptoReset
```

The OnCryptoReset event is fired when the engine needs to reset the block ciper data after encrypting a buffer. Please see the Encryption topic for more information on the default block cipher encryption in DBISAM.

Note
If you assign an event handler to this event to override the default encryption reset, you should make sure that you also assign an event handler to the OnCryptoInit, OnEncryptBlock, OnDecryptBlock events. Failure to do so can cause serious problems such as access violations.

TDBISAMEngine.OnCustomFunction Event

```
__property TCustomFunctionEvent OnCustomFunction
```

The OnCustomFunction event is fired when the engine has encountered a function name that it does not recognize as a standard function in a filter expression or SQL statement. Assigning an event handler to this event allows for the implementation of custom functions that can be used in filter expressions or SQL statements.

Note
If an event handler is not assigned to this event or the event handler doesn't assign a value to the Result variant parameter, DBISAM will treat the function as if it returned a NULL.

TDBISAMEngine.OnDecompress Event

```
__property TDecompressEvent OnDecompress
```

The OnDecompress event is fired when the engine needs to decompress a buffer previously compressed using an OnCompress event handler. Please see the Compression topic for more information on the default compression in DBISAM.

Note
If you assign an event handler to this event to override the default decompression, you should make sure that you also assign an event handler to the OnCompress event. Failure to do so can cause serious problems such as access violations.

TDBISAMEngine.OnDecryptBlock Event

```
__property TDecryptBlockEvent OnDecryptBlock
```

The OnDecryptBlock event is fired when the engine needs to decrypt an 8-byte block of data. Please see the Encryption topic for more information on the default block cipher encryption in DBISAM.

Note
If you assign an event handler to this event to override the default block decryption, you should make sure that you also assign an event handler to the OnCryptoInit, OnCryptoReset, and OnEncryptBlock events. Failure to do so can cause serious problems such as access violations.

TDBISAMEngine.OnDeleteError Event

```
__property TErrorEvent OnDeleteError
```

The OnDeleteError event is fired whenever the engine encounters an error during the deletion of a record in a table. Assigning an event handler to this event allows you to retry, abort, or fail any delete operation on any record in any table.

Note
You cannot modify the contents of the CurrentRecord parameter in this event. Also, this event is not triggered during any system processing such as Creating and Altering Tables or Optimizing Tables.

TDBISAMEngine.OnEncryptBlock Event

```
__property TEncryptBlockEvent OnEncryptBlock
```

The OnEncryptBlock event is fired when the engine needs to encrypt an 8-byte block of data. Please see the Encryption topic for more information on the default block cipher encryption in DBISAM.

Note
If you assign an event handler to this event to override the default block encryption, you should make sure that you also assign an event handler to the OnCryptoInit, OnCryptoReset, and OnDecryptBlock events. Failure to do so can cause serious problems such as access violations.

TDBISAMEngine.OnInsertError Event

```
__property TErrorEvent OnInsertError
```

The OnInsertError event is fired whenever the engine encounters an error during the insertion of a record in a table. Assigning an event handler to this event allows you to retry, abort, or fail any insert operation on any record in any table.

Note
You cannot modify the contents of the CurrentRecord parameter in this event. Also, this event is not triggered during any system processing such as Creating and Altering Tables or Optimizing Tables.

TDBISAMEngine.OnServerConnect Event

```
__property TServerConnectEvent OnServerConnect
```

The OnServerConnect event is fired when the EngineType property is set to etServer and the database server has accepted a connection from a remote session. The UserData variable parameter allows the association of a user-defined object, such as a TListItem object in a TListView component, with the session until it is removed on the database server.

TDBISAMEngine.OnServerDisconnect Event

```
__property TServerDisconnectEvent OnServerDisconnect
```

The OnServerDisconnect event is fired when the EngineType property is set to etServer and a remote session disconnects from the database server. The UserData parameter is the same object reference that was originally assigned to the connection in an OnServerConnect event handler.

TDBISAMEngine.OnServerLogCount Event

```
__property TServerLogCountEvent OnServerLogCount
```

The OnServerLogCount event is fired when the EngineType property is set to etServer and the database server needs to retrieve the total number of logs in the database server log. Log records are added to the database server log via an OnServerLogEvent event handler.

TDBISAMEngine.OnServerLogEvent Event

```
__property TServerLogEvent OnServerLogEvent
```

The OnServerLogEvent event is fired when the EngineType property is set to etServer and the database server needs to log an event that has occurred. An event can be anything from a remote session connection being initiated or destroyed to a general or severe error with the database server.

TDBISAMEngine.OnServerLogin Event

```
__property TServerLoginEvent OnServerLogin
```

The OnServerLogin event is fired when the EngineType property is set to etServer and a remote session logs in to the database server. The UserData parameter is the same object reference that was originally assigned to the connection in an OnServerConnect event handler.

TDBISAMEngine.OnServerLogout Event

```
__property TServerLogoutEvent OnServerLogout
```

The OnServerLogout event is fired when the EngineType property is set to etServer and a remote session logs out from the database server. The UserData parameter is the same object reference that was originally assigned to the connection in an OnServerConnect event handler.

TDBISAMEngine.OnServerLogRecord Event

```
__property TServerLogRecordEvent OnServerLogRecord
```

The OnServerLogRecord event is fired when the EngineType property is set to etServer and the database server needs to retrieve specific log record in the database server log. Log records are added to the database server log via an OnServerLogEvent event handler.

TDBISAMEngine.OnServerProcedure Event

```
__property TServerProcedureEvent OnServerProcedure
```

The OnServerProcedure event is fired when the EngineType property is set to etServer and a remote session calls a server-side procedure. Assigning an event handler to this event allows for the implementation of server-side procedures that can be called by remote sessions to perform any kind of server-side processing.

Note
If an event handler is not assigned to this event or the event handler doesn't modify the parameters passed to the server-side procedure, DBISAM will return the same parameters passed to the server-side procedure back to the remote session.

TDBISAMEngine.OnServerReconnect Event

```
__property TServerReconnectEvent OnServerReconnect
```

The OnServerReconnect event is fired when the EngineType property is set to etServer and a remote session reconnects to the database server. The UserData parameter is the same object reference that was originally assigned to the connection in an OnServerConnect event handler.

TDBISAMEngine.OnServerScheduledEvent Event

```
__property TServerScheduledEvent OnServerScheduledEvent
```

The OnServerScheduledEvent event is fired when the EngineType property is set to etServer and a scheduled event is launched by the database server. Assigning an event handler to this event allows for the implementation of scheduled events like online backups on the database server.

Note
If an event handler is not assigned to this event or the event handler doesn't set the Completed variable parameter to True, DBISAM will keep attempting to launch the scheduled event until the Completed variable parameter is set to True or the time period configured for the scheduled event elapses.

TDBISAMEngine.OnServerStart Event

```
__property System::Classes::TNotifyEvent OnServerStart
```

The OnServerStart event is fired when the EngineType property is set to etServer the database server is started by setting the Active property to True.

TDBISAMEngine.OnServerStop Event

```
__property System::Classes::TNotifyEvent OnServerStop
```

The OnServerStop event is fired when the EngineType property is set to etServer the database server is stopped by setting the Active property to False.

TDBISAMEngine.OnShutdown Event

```
__property System::Classes::TNotifyEvent OnShutdown
```

The OnShutdown event is fired when the DBISAM engine is shut down. Assign an event handler to the OnShutdown event to take specific actions when an application deactivates the engine. The engine is deactivated by setting its Active property to False.

Note
You should not toggle the Active property from within this event handler. Doing so can cause infinite recursion.

TDBISAMEngine.OnStartup Event

```
__property System::Classes::TNotifyEvent OnStartup
```

The OnStartup event is fired when the DBISAM engine is started. Assign an event handler to the OnStartup event to take specific actions when an application activates the engine. The engine is activated by setting its Active property to True or by opening or activating a TDBISAMSession, TDBISAMDatabase, TDBISAMQuery, or TDBISAMTable component.

Note
You should not toggle the Active property from within this event handler. Doing so can cause infinite recursion.

TDBISAMEngine.OnTextIndexFilter Event

```
__property TTextIndexFilterEvent OnTextIndexFilter
```

The OnTextIndexFilter event is fired when the engine needs to parse a string or memo field into the appropriate words for use in full text indexing. Assigning an event handler to this event allows you to modify the string or memo field prior to the parsing taking place. This is useful when the text being indexed is in a document format like HTML or RTF and it needs to be stripped of any control or formatting codes before it can be indexed properly.

TDBISAMEngine.OnTextIndexTokenFilter Event

```
__property TTextIndexTokenFilterEvent OnTextIndexTokenFilter
```

The OnTextIndexTokenFilter event is fired after the engine has parsed a string or memo field into words for full text indexing but before it actually performs the indexing. Assigning an event handler to this event allows you to filter out any undesired words before they are added to the full text index.

TDBISAMEngine.OnUpdateError Event

```
__property TErrorEvent OnUpdateError
```

The OnUpdateError event is fired whenever the engine encounters an error during the update of a record in a table. Assigning an event handler to this event allows you to retry, abort, or fail any update operation on any record in any table.

Note
You cannot modify the contents of the CurrentRecord parameter in this event. Also, this event is not triggered during any system processing such as Creating and Altering Tables or Optimizing Tables.

TDBISAMEngine.RecordLockTrigger Event

```
__property TRecordLockTriggerEvent RecordLockTrigger
```

The RecordLockTrigger event is fired after the engine has locked a record in a table. Assigning an event handler to this event allows you to perform processing after the record is locked.

TDBISAMEngine.RecordUnlockTrigger Event

```
__property TRecordLockTriggerEvent RecordUnlockTrigger
```

The RecordUnlockTrigger event is fired after the engine has unlocked a record in a table. Assigning an event handler to this event allows you to perform processing after the record is unlocked.

TDBISAMEngine.RollbackTrigger Event

```
__property TEndTransactionTriggerEvent RollbackTrigger
```

The RollbackTrigger event is fired after the engine has rolled back a transaction for a database. Assigning an event handler to this event allows you to perform processing after the transaction rollback.

TDBISAMEngine.SQLTrigger Event

```
__property TSQLTriggerEvent SQLTrigger
```

The SQLTrigger event is fired after the engine has executed an SQL statement for a database. Assigning an event handler to this event allows you to perform processing after the SQL execution, such as logging the SQL that was executed.

TDBISAMEngine.StartTransactionTrigger Event

```
__property TStartTransactionTriggerEvent StartTransactionTrigger
```

The StartTransactionTrigger event is fired after the engine has started a transaction for a database. Assigning an event handler to this event allows you to perform processing after the transaction start.

5.9 TDBISAMFieldDef Component

Header File: dbisamtb

Inherits From Db

Use the TDBISAMFieldDef object to access a field definition for a table when reading the structure information for a table or to define a field definition for a table when creating or altering the structure of a table using the TDBISAMTable CreateTable or AlterTable method.

Properties	Methods	Events
Attributes	Assign	
CharCase	AssignTo	
Compression	CreateField	
DataType	TDBISAMFieldDef	
DefaultValue		
Description		
FieldClass		
FieldNo		
MaxValue		
MinValue		
Required		
Size		

TDBISAMFieldDef.Attributes Property

```
__property Data::Db::TFieldAttributes Attributes
```

Use the Attributes property to access or define the attributes for the field definition.

TDBISAMFieldDef.CharCase Property

```
__property TFieldCharCase CharCase
```

Use the CharCase property to access or define the character-casing for the field definition.

TDBISAMFieldDef.Compression Property

```
__property System::Byte Compression
```

Use the Compression property to access or define the compression for the field definition. The compression is specified as a Byte value between 0 and 9, with the default being 0, or none, and 6 being the best selection for size/speed. The default compression is ZLib, but can be replaced by using the TDBISAMEngine events for specifying a different type of compression. Please see the Compression and Customizing the Engine topics for more information.

TDBISAMFieldDef.DataType Property

```
__property Data::Db::TFieldType DataType
```

Use the DataType property to access or define the data type for the field definition.

TDBISAMFieldDef.DefaultValue Property

```
__property System::UnicodeString DefaultValue
```

Use the DefaultValue property to access or define the default value for the field definition. The expression used for the default value must be a simple literal constant or one of the following SQL functions:

CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_GUID

TDBISAMFieldDef.Description Property

```
__property System::UnicodeString Description
```

Use the Description property to access or define the description for the field definition.

TDBISAMFieldDef.FieldClass Property

```
__property Data::Db::TFieldClass FieldClass
```

This property is used internally by DBISAM for creating TField compoonents from a TDBISAMFieldDef object.

TDBISAMFieldDef.FieldNo Property

```
__property int FieldNo
```

Use the DefaultValue property to access or define the field number for the field definition.

Note
The field number should only be specifically defined when altering the structure of a table using the TDBISAMTable AlterTable method.

TDBISAMFieldDef.MaxValue Property

```
__property System::UnicodeString MaxValue
```

Use the MaxValue property to access or define the maximum value for the field definition. The expression used for the default value must be a simple literal constant or one of the following SQL functions:

CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_GUID

TDBISAMFieldDef.MinValue Property

```
__property System::UnicodeString MinValue
```

Use the MinValue property to access or define the minimum value for the field definition. The expression used for the default value must be a simple literal constant or one of the following SQL functions:

CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_GUID

TDBISAMFieldDef.Required Property

```
__property bool Required
```

Use the Required property to access or define whether the field should be required in the field definition.

TDBISAMFieldDef.Size Property

```
__property int Size
```

Use the Size property to access or define the size for the field definition. Only field definitions set to use the following data types should have their Size property set:

ftString
ftFixedChar
ftGuid
ftBytes
ftVarBytes
ftBCD

Note
Field definitions set to use the ftBCD type use the Size property to determine the maximum number of decimal places to use for the field definition. DBISAM allows a maximum of 4 decimal places for BCD fields.

TDBISAMFieldDef.Assign Method

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

Call the Assign method to assign another TDBISAMFieldDef or TFieldDef object to the current TDBISAMFieldDef object. The TDBISAMFieldDef object is assignment-compatible with the TFieldDef object.

TDBISAMFieldDef.AssignTo Method

```
virtual void __fastcall AssignTo(System::Classes::TPersistent*
      Dest)
```

Call the AssignTo method to assign the current TDBISAMFieldDef object to another TDBISAMFieldDef or TFieldDef object. The TDBISAMFieldDef object is assignment-compatible with the TFieldDef object.

TDBISAMFieldDef.CreateField Method

```
Data::Db::TField* __fastcall
      CreateField(System::Classes::TComponent* Owner,
      Data::Db::TObjectField* ParentField =
      (Data::Db::TObjectField*)(0x0), const System::UnicodeString
      FieldName = System::UnicodeString(), bool CreateChildren = true)
```

Call the CreateField method to create an appropriate TField descendant component for the current field definition.

The only necessary parameter to this method is the Owner parameter, which is usually the form or data module that is creating the field components. The rest of the parameters can be left as the default values.

TDBISAMFieldDef.TDBISAMFieldDef Method

```
__fastcall TDBISAMFieldDef(TDBISAMFieldDefs* Owner, const
      System::UnicodeString Name, Data::Db::TFieldType DataType, int
      Size, bool Required, const System::UnicodeString DefaultValue,
      const System::UnicodeString MinValue, const
      System::UnicodeString MaxValue, const System::UnicodeString
      Description, TFieldCharCase CharCase, System::Byte Compression,
      int FieldNo)
```

Use the New operator to create an instance of a TDBISAMFieldDef object using the constructor. However, you should never need to create an instance of the TDBISAMFieldDef object since one is automatically created when using the various methods of the TDBISAMFieldDefs object.

5.10 TDBISAMFieldDefs Component

Header File: dbisamtb

Inherits From Db

Use the TDBISAMFieldDefs object to access the field definitions for a table when reading the structure information for a table or to define the field definitions for a table when creating or altering the structure of a table using the TDBISAMTable CreateTable or AlterTable method.

Properties	Methods	Events
Items	Add	
	AddFieldDef	
	Find	
	Insert	
	InsertFieldDef	
	TDBISAMFieldDefs	
	Update	

TDBISAMFieldDefs.Items Property

```
__property TDBISAMFieldDef* Items[int Index]
```

Use the Items property to access a specific field definition. The Index is an integer identifying the field definition's position in the list of field definitions, in the range of 0 to the value of the Count property minus 1.

TDBISAMFieldDefs.Add Method

```
HIDESBASE void __fastcall Add(const System::UnicodeString Name,
      Data::Db::TFieldType DataType, int Size = 0x0, bool Required =
      false, const System::UnicodeString DefaultValue =
      System::UnicodeString(), const System::UnicodeString MinValue =
      System::UnicodeString(), const System::UnicodeString MaxValue =
      System::UnicodeString(), const System::UnicodeString Description
      = System::UnicodeString(), TFieldCharCase CharCase =
      (TFieldCharCase)(0x0), System::Byte Compression =
      (System::Byte)(0x0))

HIDESBASE void __fastcall Add(int FieldNo, const
      System::UnicodeString Name, Data::Db::TFieldType DataType, int
      Size = 0x0, bool Required = false, const System::UnicodeString
      DefaultValue = System::UnicodeString(), const
      System::UnicodeString MinValue = System::UnicodeString(), const
      System::UnicodeString MaxValue = System::UnicodeString(), const
      System::UnicodeString Description = System::UnicodeString(),
      TFieldCharCase CharCase = (TFieldCharCase)(0x0), System::Byte
      Compression = (System::Byte)(0x0))
```

Call the Add method to add a new field definition object to the list of field definitions for the table. Please see the TDBISAMFieldDef object for more information on each parameter to this method.

TDBISAMFieldDefs.AddFieldDef Method

```
TDBISAMFieldDef* __fastcall AddFieldDef(void)
```

Call the AddFieldDef method to add an empty TDBISAMFieldDef field definition object to the list of field definitions. You can then use the individual properties of the TDBISAMFieldDef object that is returned to define the properties of the field.

TDBISAMFieldDefs.Find Method

```
HIDESBASE TDBISAMFieldDef* __fastcall Find(const
      System::UnicodeString Name)
```

Call the Find method to locate a specific field definition by the name of the field. If a field definition is found with the same name, a reference to the field definition is returned, otherwise an exception is raised. The search against the field names is case-insensitive.

TDBISAMFieldDefs.Insert Method

```
HIDESBASE void __fastcall Insert(int InsertPos, const
      System::UnicodeString Name, Data::Db::TFieldType DataType, int
      Size = 0x0, bool Required = false, const System::UnicodeString
      DefaultValue = System::UnicodeString(), const
      System::UnicodeString MinValue = System::UnicodeString(), const
      System::UnicodeString MaxValue = System::UnicodeString(), const
      System::UnicodeString Description = System::UnicodeString(),
      TFieldCharCase CharCase = (TFieldCharCase)(0x0), System::Byte
      Compression = (System::Byte)(0x0))

HIDESBASE void __fastcall Insert(int InsertPos, int FieldNo,
      const System::UnicodeString Name, Data::Db::TFieldType DataType,
      int Size = 0x0, bool Required = false, const
      System::UnicodeString DefaultValue = System::UnicodeString(),
      const System::UnicodeString MinValue = System::UnicodeString(),
      const System::UnicodeString MaxValue = System::UnicodeString(),
      const System::UnicodeString Description =
      System::UnicodeString(), TFieldCharCase CharCase =
      (TFieldCharCase)(0x0), System::Byte Compression =
      (System::Byte)(0x0))
```

Call the Insert method to insert a new field definition object at a specific position in the list of field definitions for the table. The InsertPos parameter is an integer identifying the field definition's position in the list of field definitions, in the range of 0 to the value of the Count property minus 1. Please see the TDBISAMFieldDef object for more information on each parameter to this method.

TDBISAMFieldDefs.InsertFieldDef Method

```
TDBISAMFieldDef* __fastcall InsertFieldDef(int InsertPos)
```

Call the InsertFieldDef method to insert an empty TDBISAMFieldDef field definition object at a specific position in the list of field definitions. You can then use the individual properties of the TDBISAMFieldDef object that is returned to define the properties of the field. The InsertPos parameter is an integer identifying the field definition's position in the list of field definitions, in the range of 0 to the value of the Count property minus 1.

TDBISAMFieldDefs.TDBISAMFieldDefs Method

```
__fastcall TDBISAMFieldDefs(System::Classes::TPersistent*
      AOwner)
```

Use the New operator to create an instance of a TDBISAMFieldDefs object using the constructor. However, you should never need to create an instance of the TDBISAMFieldDefs object since one is automatically created for every table when the TDBISAMTable component is created.

TDBISAMFieldDefs.Update Method

```
HIDESBASE void __fastcall Update(void)
```

Call the Update method to retrieve the current field definitions from the table. Calling this method will clear the existing field definitions in the list of field definitions and replace them the field definitions from the table specified by the owner TDBISAMTable DatabaseName and TableName properties.

5.11 TDBISAMFunction Component

Header File: dbisamtb

Inherits From Classes

Use the TDBISAMFunction object to access or define a custom SQL or filter function in the TDBISAMEngine component. Please see the Customizing the Engine topic for more information.

Properties	Methods	Events
Name	Assign	
Params	TDBISAMFunction	
ResultType		

TDBISAMFunction.Name Property

```
__property System::UnicodeString Name
```

Use the Name property to specify the name of the custom function. This name is case-insensitive when referred to in filter expressions or SQL statements.

TDBISAMFunction.Params Property

```
__property TDBISAMFunctionParams* Params
```

Use the Params property to define the parameters to the custom function using the CreateFunctionParam method.

TDBISAMFunction.ResultType Property

```
__property Data::Db::TFieldType ResultType
```

Use the ResultType property to define the resultant data type of the custom function.

TDBISAMFunction.Assign Method

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

Call the Assign method to assign another TDBISAMFunction object to the current TDBISAMFunction object.

TDBISAMFunction.TDBISAMFunction Method

```
__fastcall virtual TDBISAMFunction(System::Classes::TCollection*
      Collection)
```

Use the New operator to create an instance of a TDBISAMFunction object using the constructor. However, you should never need to create an instance of the TDBISAMFunction object since one is automatically created when using the CreateFunction method of the TDBISAMFunctions object.

5.12 TDBISAMFunctionParam Component

Header File: dbisamtb

Inherits From Classes

Use the TDBISAMFunctionParam object to access or define a parameter to a custom SQL or filter function in the TDBISAMEngine component. Please see the Customizing the Engine topic for more information.

Properties	Methods	Events
DataType	Assign	
	TDBISAMFunctionParam	

TDBISAMFunctionParam.DataType Property

```
__property Data::Db::TFieldType DataType
```

Use the DataType property to specify the data type of the parameter to the custom function.

TDBISAMFunctionParam.Assign Method

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

Call the Assign method to assign another TDBISAMFunctionParam object to the current TDBISAMFunctionParam object.

TDBISAMFunctionParam.TDBISAMFunctionParam Method

```
__fastcall virtual
      TDBISAMFunctionParam(System::Classes::TCollection* Collection)

__fastcall TDBISAMFunctionParam(TDBISAMFunctionParams*
      AFunctionParams)
```

Use the New operator to create an instance of a TDBISAMFunctionParam object using the constructor. However, you should never need to create an instance of the TDBISAMFunctionParam object since one is automatically created when using the CreateFunctionParam method of the TDBISAMFunctionParams object.

5.13 TDBISAMFunctionParams Component

Header File: dbisamtb

Inherits From Classes

Use the TDBISAMFunctionParams object to access or define the parameters to a custom SQL or filter function in the TDBISAMEngine component. Please see the Customizing the Engine topic for more information.

Properties	Methods	Events
Items	Assign	
	CreateFunctionParam	
	IsEqual	
	TDBISAMFunctionParams	

TDBISAMFunctionParams.Items Property

```
__property TDBISAMFunctionParam* Items[int Index]
```

Use the Items property to access a specific function parameter definition. The Index is an integer identifying the parameter definition's position in the list of parameter definitions, in the range of 0 to the value of the Count property minus 1.

TDBISAMFunctionParams.Assign Method

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

Call the Assign method to assign another TDBISAMFunctionParams object to the current TDBISAMFunctionParams object.

TDBISAMFunctionParams.CreateFunctionParam Method

```
TDBISAMFunctionParam* __fastcall
      CreateFunctionParam(Data::Db::TFieldType DataType)
```

Call the CreateFunctionParam method to create a new custom function parameter definition and return a reference to the parameter definition. The DataType parameter specifies the data type of the parameter.

TDBISAMFunctionParams.IsEqual Method

```
bool __fastcall IsEqual(TDBISAMFunctionParams* Value)
```

The IsEqual method compares the current custom function parameter definitions against another list of parameter definitions and returns True if they are equal and False if they are not.

TDBISAMFunctionParams.TDBISAMFunctionParams Method

```
__fastcall TDBISAMFunctionParams(System::Classes::TPersistent*
      Owner)
```

Use the New operator to create an instance of a TDBISAMFunctionParams object using the constructor. However, you should never need to create an instance of the TDBISAMFunctionParams object since one is automatically created when using the CreateFunction method of the TDBISAMFunctions object.

5.14 TDBISAMFunctions Component

Header File: dbisamtb

Inherits From Classes

Use the TDBISAMFunctions object to access or define the custom SQL or filter functions in the TDBISAMEngine component. Please see the Customizing the Engine topic for more information.

Properties	Methods	Events
Items	Assign	
	CreateFunction	
	FindFunction	
	FunctionByName	
	IsEqual	
	TDBISAMFunctions	

TDBISAMFunctions.Items Property

```
__property TDBISAMFunction* Items[int Index]
```

Use the Items property to access a specific custom function definition. The Index is an integer identifying the custom function definition's position in the list of function definitions, in the range of 0 to the value of the Count property minus 1.

TDBISAMFunctions.Assign Method

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

Call the Assign method to assign another TDBISAMFunctions object to the current TDBISAMFunctions object.

TDBISAMFunctions.CreateFunction Method

```
TDBISAMFunction* __fastcall CreateFunction(Data::Db::TFieldType
      ResultType, const System::UnicodeString FunctionName)
```

Call the CreateFunction method to create a new custom function definition and return a reference to the function definition. The ResultType parameter specifies the resultant data type and the FunctionName parameter specifies the name of the custom function. This name is case-insensitive when referred to in filter expressions or SQL statements.

TDBISAMFunctions.FindFunction Method

```
TDBISAMFunction* __fastcall FindFunction(const
      System::UnicodeString Value)
```

Call the FindFunction method to locate a specific custom function definition by the name of the custom function. If a custom function definition is found with the same name, a reference to the function definition is returned, otherwise nil is returned. The search against the function names is case-insensitive.

TDBISAMFunctions.FunctionByName Method

```
TDBISAMFunction* __fastcall FunctionByName(const
      System::UnicodeString Value)
```

Call the FunctionByName method to locate a specific custom function definition by the name of the custom function. If a custom function definition is found with the same name, a reference to the function definition is returned, otherwise an exception is raised. The search against the function names is case-insensitive. The FunctionByName method is essentially a wrapper around the FindFunction method that raises an exception if the FindFunction method returns nil.

TDBISAMFunctions.IsEqual Method

```
bool __fastcall IsEqual(TDBISAMFunctions* Value)
```

The IsEqual method compares the current custom function definitions against another list of custom function definitions and returns True if they are equal and False if they are not.

TDBISAMFunctions.TDBISAMFunctions Method

```
__fastcall TDBISAMFunctions(System::Classes::TComponent* Owner)
```

Use the New operator to create an instance of a TDBISAMFunctions object using the constructor. However, you should never need to create an instance of the TDBISAMFunctions object since one is automatically created when the TDBISAMEngine component is created.

5.15 TDBISAMIndexDef Component

Header File: dbisamtb

Inherits From Db

Use the TDBISAMIndexDef object to access an index definition for a table when reading the structure information for a table or to define an index definition for a table when creating or altering the structure of a table using the TDBISAMTable CreateTable or AlterTable method.

Properties	Methods	Events
Compression	Assign	
DescFields	AssignTo	
FieldExpression	TDBISAMIndexDef	
Fields		
NoKeyStatistics		
Options		

TDBISAMIndexDef.Compression Property

```
__property TIndexCompression Compression
```

Use the Compression property to access or define the compression for the index definition. For more information please see the Index Compression topic.

TDBISAMIndexDef.DescFields Property

```
__property System::UnicodeString DescFields
```

Use the DescFields property to access or define the descending fields for the index definition. Multiple field names should be separated with a semicolon (;).

TDBISAMIndexDef.FieldExpression Property

```
__property System::UnicodeString FieldExpression
```

This property will always be the same value as the FieldExpression property and should be ignored.

TDBISAMIndexDef.Fields Property

```
__property System::UnicodeString Fields
```

Use the Fields property to access or define the fields being indexed in the index definition. Multiple field names should be separated with a semicolon (;).

TDBISAMIndexDef.NoKeyStatistics Property

```
__property bool NoKeyStatistics
```

Use the NoKeyStatistics property to set whether index statistics will be used for the index definition. Under most circumstances you should leave this property set to the default of False. Not using the index statistics is only useful for very large tables where insert/update/delete performance is very important, and where it is acceptable to not have logical record numbers or statistics for optimizing filters and queries.

TDBISAMIndexDef.Options Property

```
__property Data::Db::TIndexOptions Options
```

Use the Options property to access or define the indexing options for the index definition.

TDBISAMIndexDef.Assign Method

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

Call the Assign method to assign another TDBISAMIndexDef or TIndexDef object to the current TDBISAMIndexDef object. The TDBISAMIndexDef object is assignment-compatible with the TIndexDef object.

TDBISAMIndexDef.AssignTo Method

```
virtual void __fastcall AssignTo(System::Classes::TPersistent*
      Dest)
```

Call the AssignTo method to assign the current TDBISAMIndexDef object to another TDBISAMIndexDef or TIndexDef object. The TDBISAMIndexDef object is assignment-compatible with the TIndexDef object.

TDBISAMIndexDef.TDBISAMIndexDef Method

```
__fastcall TDBISAMIndexDef(TDBISAMIndexDefs* Owner, const
      System::UnicodeString Name, const System::UnicodeString Fields,
      Data::Db::TIndexOptions Options, const System::UnicodeString
      DescFields, TIndexCompression Compression, bool NoKeyStats)
```

Use the New operator to create an instance of a TDBISAMIndexDef object using the constructor. However, you should never need to create an instance of the TDBISAMIndexDef object since one is automatically created when using the various methods of the TDBISAMIndexDefs object.

5.16 TDBISAMIndexDefs Component

Header File: dbisamtb

Inherits From Db

Use the TDBISAMIndexDefs object to access the index definitions for a table when reading the structure information for a table or to define the index definitions for a table when creating or altering the structure of a table using the TDBISAMTable CreateTable or AlterTable method.

Properties	Methods	Events
Items	Add	
	AddIndexDef	
	Find	
	FindIndexForFields	
	GetIndexForFields	
	TDBISAMIndexDefs	
	Update	

TDBISAMIndexDefs.Items Property

```
__property TDBISAMIndexDef* Items[int Index]
```

Use the Items property to access a specific index definition. The Index is an integer identifying the index definition's position in the list of index definitions, in the range of 0 to the value of the Count property minus 1.

TDBISAMIndexDefs.Add Method

```
HIDESBASE void __fastcall Add(const System::UnicodeString Name,
      const System::UnicodeString Fields, Data::Db::TIndexOptions
      Options = Data::Db::TIndexOptions() , const
      System::UnicodeString DescFields = System::UnicodeString(),
      TIndexCompression Compression = (TIndexCompression)(0x0), bool
      NoKeyStatistics = false)
```

Call the Add method to add a new index definition object to the list of index definitions for the table. Please see the TDBISAMIndexDef object for more information on each parameter to this method.

TDBISAMIndexDefs.AddIndexDef Method

```
TDBISAMIndexDef* __fastcall AddIndexDef(void)
```

Call the AddIndexDef method to add an empty TDBISAMIndexDef index definition object to the list of index definitions. You can then use the individual properties of the TDBISAMIndexDef object that is returned to define the properties of the field.

TDBISAMIndexDefs.Find Method

```
HIDESBASE TDBISAMIndexDef* __fastcall Find(const
      System::UnicodeString Name)
```

Call the Find method to locate a specific index definition by the name of the index. If an index definition is found with the same name, a reference to the index definition is returned, otherwise an exception is raised. The search against the index names is case-insensitive.

Note
To search for the primary index definition just use a blank string ("").

TDBISAMIndexDefs.FindIndexForFields Method

```
TDBISAMIndexDef* __fastcall FindIndexForFields(const
      System::UnicodeString Fields)
```

Call the FindIndexForFields method to locate a specific index definition by the fields that make up the index. The Fields parameter specifies the fields to use in the search. If multiple field names are specified, separate each field name with a semicolon (;). If an index definition is found that matches the specified fields, a reference to the index definition is returned, otherwise an exception is raised. The search against the field names is case-insensitive. The FindIndexForFields method is essentially a wrapper around the GetIndexForFields method that raises an exception if the GetIndexForFields method returns nil.

TDBISAMIndexDefs.GetIndexForFields Method

```
TDBISAMIndexDef* __fastcall GetIndexForFields(const
      System::UnicodeString Fields, bool CaseInsensitive)
```

Call the GetIndexForFields method to locate a specific index definition by the fields that make up the index. The Fields parameter specifies the fields to use in the search. If multiple field names are specified, separate each field name with a semicolon (;). If an index definition is found that matches the specified fields, a reference to the index definition is returned, otherwise nil is returned. The search against the field names is case-insensitive.

TDBISAMIndexDefs.TDBISAMIndexDefs Method

```
__fastcall TDBISAMIndexDefs(System::Classes::TPersistent*
      AOwner)
```

Use the New operator to create an instance of a TDBISAMIndexDefs object using the constructor. However, you should never need to create an instance of the TDBISAMIndexDefs object since one is automatically created for every table when the TDBISAMTable component is created.

TDBISAMIndexDefs.Update Method

```
HIDESBASE void __fastcall Update(void)
```

Call the Update method to retrieve the current index definitions from the table. Calling this method will clear the existing index definitions in the list of index definitions and replace them the index definitions from the table specified by the owner TDBISAMTable DatabaseName and TableName properties.

5.17 TDBISAMParam Component

Header File: dbisamtb

Inherits From Classes

Use the TDBISAMParam object to access or define a parameter to an SQL statement in the TDBISAMQuery component or a server-side procedure in the TDBISAMSession and TDBISAMEnginecomponents. Please see the Parameterized Queries, Calling Server-Side Procedures, and Customizing the Engine topics for more information.

Properties	Methods	Events
AsBCD	Assign	
AsBlob	AssignField	
AsBoolean	AssignFieldValue	
AsCurrency	Clear	
AsDate	GetData	
AsDateTime	GetDataSize	
AsFloat	LoadFromFile	
AsInteger	LoadFromStream	
AsLargeInt	SaveToFile	
AsMemo	SaveToStream	
AsSmallInt	SetBlobData	
AsString	SetData	
AsTime	TDBISAMParam	
AsWord		
Bound		
DataType		
IsNull		
Name		
Text		
Value		

TDBISAMParam.AsBCD Property

```
__property System::Currency AsBCD
```

Use the AsBCD property to access or specify the current value of the parameter as a Currency type. Setting the value of the parameter using the AsBCD method will set the DataType property to ftBCD.

TDBISAMParam.AsBlob Property

```
__property System::RawByteString AsBlob
```

Use the AsBlob property to access or specify the current value of the parameter as a String type. Setting the value of the parameter using the AsBlob method will set the DataType property to ftBlob.

TDBISAMParam.AsBoolean Property

```
__property bool AsBoolean
```

Use the AsBoolean property to access or specify the current value of the parameter as a Boolean type. Setting the value of the parameter using the AsBoolean method will set the DataType property to ftBoolean.

TDBISAMParam.AsCurrency Property

```
__property System::Currency AsCurrency
```

Use the AsCurrency property to access or specify the current value of the parameter as a Currency type. Setting the value of the parameter using the AsCurrency method will set the DataType property to ftCurrency.

TDBISAMParam.AsDate Property

```
__property System::TDateTime AsDate
```

Use the AsDate property to access or specify the current value of the parameter as a TDateTime type. Setting the value of the parameter using the AsDate method will set the DataType property to ftDate.

TDBISAMParam.AsDateTime Property

```
__property System::TDateTime AsDateTime
```

Use the AsDateTime property to access or specify the current value of the parameter as a TDateTime type. Setting the value of the parameter using the AsDateTime method will set the DataType property to ftDateTime.

TDBISAMParam.AsFloat Property

```
__property double AsFloat
```

Use the AsFloat property to access or specify the current value of the parameter as a Double type. Setting the value of the parameter using the AsFloat method will set the DataType property to ftFloat.

TDBISAMParam.AsInteger Property

```
__property int AsInteger
```

Use the AsInteger property to access or specify the current value of the parameter as an Integer type. Setting the value of the parameter using the AsInteger method will set the DataType property to ftInteger.

TDBISAMParam.AsLargeInt Property

```
__property __int64 AsLargeInt
```

Use the AsLargeInt property to access or specify the current value of the parameter as an LargeInt type. Setting the value of the parameter using the AsLargeInt method will set the DataType property to ftLargeInt.

TDBISAMParam.AsMemo Property

```
__property System::UnicodeString AsMemo
```

Use the AsMemo property to access or specify the current value of the parameter as a String type. Setting the value of the parameter using the AsMemo method will set the DataType property to ftMemo.

TDBISAMParam.AsSmallInt Property

```
__property int AsSmallInt
```

Use the AsSmallInt property to access or specify the current value of the parameter as a SmallInt type. Setting the value of the parameter using the AsSmallInt method will set the DataType property to ftSmallInt.

TDBISAMParam.AsString Property

```
__property System::UnicodeString AsString
```

Use the AsString property to access or specify the current value of the parameter as a String type. Setting the value of the parameter using the AsString method will set the DataType property to ftString.

TDBISAMParam.AsTime Property

```
__property System::TDateTime AsTime
```

Use the AsTime property to access or specify the current value of the parameter as a TDateTime type. Setting the value of the parameter using the AsTime method will set the DataType property to ftTime.

TDBISAMParam.AsWord Property

```
__property int AsWord
```

Use the AsWord property to access or specify the current value of the parameter as a Word type. Setting the value of the parameter using the AsWord method will set the DataType property to ftWord.

TDBISAMParam.Bound Property

```
__property bool Bound
```

Use the Bound property to determine whether a value has been assigned to the parameter. Whenever a value is assigned to the TDBISAMParam object, the Bound property is automatically set to True. Set the Bound property to False to undo the setting of the value. The Clear method will replace the value of the parameter with NULL, but will not set the Bound property to False. If the Clear method is used to bind the parameter to a NULL value, the Bound property must be separately set to True.

TDBISAMQuery components use the value of the Bound property to determine whether or not to assign a default value for the parameter. If the Bound property is False, a TDBISAMQuery component attempts to assign a value from the data source indicated by the DataSource property of the TDBISAMQuery component.

Note
The Bound property is ignored when the TDBISAMParam object is used with server-side procedures.

TDBISAMParam.DataType Property

```
__property Data::Db::TFieldType DataType
```

Use the DataType property to discover the type of data that was assigned to the parameter. The DataType property is set automatically when a value is assigned to the parameter. Do not set the DataType property for bound fields (Bound=True), as that may cause the assigned value to be misinterpreted.

TDBISAMParam.IsNull Property

```
__property bool IsNull
```

Use the IsNull property to discover if the value of the parameter is NULL, indicating the value of a blank field. NULL values can arise in the following ways:

- Assigning the value of another, NULL, parameter.
- Assigning the value of a blank TField object using the AssignFieldValue method.
- Calling the Clear method.

Note
NULL parameters are not the same as unbound parameters. Unbound parameters have not had a value assigned. NULL parameters have a NULL value. NULL parameters may be bound or unbound.

TDBISAMParam.Name Property

```
__property System::UnicodeString Name
```

Use the Name property to specify the name for the parameter.

TDBISAMParam.Text Property

```
__property System::UnicodeString Text
```

Use the Text property to assign the value of the parameter to a string without changing the DataType property. Unlike the AsString property, which sets the value to a string and changes the DataType property, setting the Text property converts the string to the current data type of the parameter, and sets the value accordingly. Thus, use the AsString property to treat the parameter as representing the value of a string field. Use the Text property instead when assigning a value that is in string form, when making no assumptions about the data type. For example, the Text property is useful for assigning user data that was input using an edit control.

TDBISAMParam.Value Property

```
__property System::Variant Value
```

Use the Value property to manipulate the value of a parameter as a Variant without needing to know the data type the parameter represents.

TDBISAMParam.Assign Method

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

Call the Assign method to assign another TDBISAMParam or TParam object to the current TDBISAMParam object. The TDBISAMParam object is assignment-compatible with the TParam object.

Note
You can assign a TDBISAMTable or TDBISAMQuery stream directly to a TDBISAMParam object using this method. This is equivalent to using the TDBISAMTable or TDBISAMQuery SaveToStream method and then the TDBISAMParam LoadFromStream method. Likewise, you can use the TDBISAMTable or TDBISAMQuery component's Assign method to directly load a stream from a TDBISAMParam object. This is equivalent to using the TDBISAMParam SaveToStream method and then the TDBISAMTable or TDBISAMQuery LoadFromStream method.

TDBISAMParam.AssignField Method

```
void __fastcall AssignField(Data::Db::TField* Field)
```

Call the AssignField method to set a parameter to represent a particular TField object. The AssignField method sets the Bound property to True.

Note
Unlike the AssignFieldValue method, the AssignField method names the parameter after the TField object as well as taking its value.

TDBISAMParam.AssignFieldValue Method

```
void __fastcall AssignFieldValue(Data::Db::TField* Field, const
      System::Variant &Value)
```

Call the AssignFieldValue method to set the Value property to the value passed as the Value parameter. The AssignFieldValue method assumes that the Value parameter represents a value from a field like the Field parameter, and assigns the DataType property accordingly. The AssignFieldValue method sets the Bound property to True.

Note
Unlike the AssignField method, the AssignFieldValue method does not name the parameter after the TField object.

TDBISAMParam.Clear Method

```
void __fastcall Clear(void)
```

Call the Clear method to assign a NULL value to a parameter. Calling Clear neither sets nor clears the Bound property. When assigning a NULL value to a parameter, set the Bound property as well as calling the Clear method.

TDBISAMParam.GetData Method

```
void __fastcall GetData(void * Buffer)
```

Call the GetData method to retrieve the value of a parameter in its native DBISAM format into a buffer. The buffer must have enough space to hold the information. Use the GetDataSize method to determine the necessary size.

TDBISAMParam.GetDataSize Method

```
int __fastcall GetDataSize(void)
```

Call the GetDataSize method to determine the number of bytes used to represent the parameter's value in its native DBISAM format.

TDBISAMParam.LoadFromFile Method

```
void __fastcall LoadFromFile(const System::UnicodeString
      FileName, Data::Db::TBlobType BlobType)
```

Call the LoadFromFile method to set the value of a BLOB parameter from a value stored in the file specified by the FileName parameter. The DataType property is set to the value passed as the BlobType parameter.

TDBISAMParam.LoadFromStream Method

```
void __fastcall LoadFromStream(System::Classes::TStream* Stream,
      Data::Db::TBlobType BlobType)
```

Call the LoadFromStream method to set the value of a BLOB parameter from a value stored in the stream specified by the Stream parameter. The DataType property is set to the value passed as the BlobType parameter.

TDBISAMParam.SaveToFile Method

```
void __fastcall SaveToFile(const System::UnicodeString FileName)
```

Call the SaveToFile method to copy the value of a BLOB parameter to the file specified by the FileName parameter.

TDBISAMParam.SaveToStream Method

```
void __fastcall SaveToStream(System::Classes::TStream* Stream)
```

Call the SaveToStream method to copy the value of a BLOB parameter to the stream specified by the Stream parameter.

TDBISAMParam.SetBlobData Method

```
void __fastcall SetBlobData(void * Buffer, int Size)
```

Call the SetBlobData method to set the value of a parameter from a buffer. The SetBlobData method copies the number of bytes specified by the Size parameter from the buffer specified by the Buffer parameter, and sets the DataType property to ftBlob.

TDBISAMParam.SetData Method

```
void __fastcall SetData(void * Buffer)
```

Call the SetData method to set the value of a parameter from a buffer that contains data in its native DBISAM format.

TDBISAMParam.TDBISAMParam Method

```
__fastcall virtual TDBISAMParam(System::Classes::TCollection*
      Collection)

__fastcall TDBISAMParam(TDBISAMParams* AParams)
```

Use the New operator to create an instance of a TDBISAMParam object using the constructor. However, you should never need to create an instance of the TDBISAMParam object since one is automatically created when using the CreateParam method of the TDBISAMParams object.

5.18 TDBISAMParams Component

Header File: dbisamtb

Inherits From Classes

Use the TDBISAMParams object to access or define parameters to an SQL statement in the TDBISAMQuery component or a server-side procedure in the TDBISAMSession and TDBISAMEnginecomponents. Please see the Parameterized Queries, Calling Server-Side Procedures, and Customizing the Engine topics for more information.

Properties	Methods	Events
Items	AssignValues	
ParamValues	CreateParam	
	FindParam	
	IsEqual	
	ParamByName	
	TDBISAMParams	

TDBISAMParams.Items Property

```
__property TDBISAMParam* Items[int Index]
```

Use the Items property to access a specific parameter. The Index is an integer identifying the parameter's position in the list of parameters, in the range of 0 to the value of the Count property minus 1.

TDBISAMParams.ParamValues Property

```
__property System::Variant ParamValues[const System::UnicodeString ParamName]
```

Use the ParamValues property to get or set the values of parameters that are identified by name. The ParamName is a string containing the names of the individual parameters of interest. If the ParamValues property is used to access more than one parameter, the names of the parameters should be separated by a semicolon (;).

Setting the ParamValues property sets the Value property for each parameter listed in the ParamName string. Specify the values as Variants, in order, in a variant array.

If ParamName includes a name that does not match any of the parameters, an exception is raised.

TDBISAMParams.AssignValues Method

```
void __fastcall AssignValues(TDBISAMParams* Value)

void __fastcall AssignValues(Data::Db::TParams* Value)
```

Call the AssignValues method to assign the values from parameters in another TDBISAMParams or TParams object. The TDBISAMParams object is assignment-compatible with the TParams object. For each parameter, the AssignValues method attempts to find a parameter with the same Name property in Value. If successful, the parameter information (type and current data) from the Value parameter is assigned to the parameter. Parameters for which no match is found are left unchanged.

TDBISAMParams.CreateParam Method

```
TDBISAMParam* __fastcall CreateParam(Data::Db::TFieldType
      FldType, const System::UnicodeString ParamName)
```

Call the CreateParam method to create a new parameter and return a reference to the parameter. The FldType parameter specifies the data type and the ParamName parameter specifies the name of the parameter. This name is case-insensitive.

TDBISAMParams.FindParam Method

```
TDBISAMParam* __fastcall FindParam(const System::UnicodeString
      Value)
```

Call the FindParam method to locate a specific parameter by the name of the parameter. If a parameter is found with the same name, a reference to the parameter is returned, otherwise nil is returned. The search against the parameter names is case-insensitive.

TDBISAMParams.IsEqual Method

```
bool __fastcall IsEqual(TDBISAMParams* Value)
```

The IsEqual method compares the current parameters against another list of parameters and returns True if they are equal and False if they are not.

TDBISAMParams.ParamByName Method

```
TDBISAMParam* __fastcall ParamByName(const System::UnicodeString
      Value)
```

Call the ParamByName method to locate a specific parameter by the name of the parameter. If a parameter is found with the same name, a reference to the parameter is returned, otherwise an exception is raised. The search against the parameter names is case-insensitive. The ParamByName method is essentially a wrapper around the FindParam method that raises an exception if the FindParam method returns nil.

TDBISAMParams.TDBISAMParams Method

```
__fastcall TDBISAMParams(System::Classes::TComponent* Owner)
```

Use the New operator to create an instance of a TDBISAMParams object using the constructor. However, you should never need to create an instance of the TDBISAMParams object since one is automatically created when the TDBISAMQuery or TDBISAMSessioncomponent is created.

5.19 TDBISAMQuery Component

Header File: dbisamtb

Inherits From TDBISAMDBDataSet

Use the TDBISAMQuery component to access or update one or more tables in a database using SQL statements. Please see the SQL Reference Overview topic for more information on the SQL support in DBISAM.

Properties	Methods	Events
DataSource	DefineProperties	AfterExecute
EngineVersion	ExecSQL	BeforeExecute
ExecutionTime	GetDetailLinkFields	OnAlterProgress
GeneratePlan	ParamByName	OnCopyProgress
LocaleID	Prepare	OnDataLost
MaxRowCount	SaveToTable	OnExportProgress
ParamCheck	TDBISAMQuery	OnGetParams
ParamCount	UnPrepare	OnImportProgress
Params		OnIndexProgress
Plan		OnLoadFromStreamProgress
Prepared		OnOptimizeProgress
ReadOnly		OnQueryError
RequestLive		OnQueryProgress
ResultIsLive		OnRepairLog
RowsAffected		OnRepairProgress
SQL		OnSaveProgress
SQLStatementType		OnSaveToStreamProgress
StmtHandle		OnSQLChanged
TableName		OnUpgradeLog
Text		OnUpgradeProgress
		OnVerifyLog
		OnVerifyProgress

TDBISAMQuery.DataSource Property

```
__property Data::Db::TDataSource* DataSource
```

The DataSource property specifies the TDataSource component from which to extract current field values to use in the identically-named parameters in the query's SQL statement specified via the SQL property. This allows you to automatically fill parameters in a query with fields values from another data source. Parameters that have the same name as fields in the other data source 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 set by the application manually.

DataSource must point to a TDataSource component linked to another dataset component; it cannot point to this TDBISAMQuery component. The dataset specified in the TDataSource component's DataSet property must be created, populated, and opened before attempting to bind parameters. Parameters are bound by calling the Prepare method prior to executing the query using the ExecSQL or Open method. If the SQL statement used by the query does not contain parameters, or all parameters are bound by the application using the Params property or the ParamByName method, the DataSource property need not be assigned.

If the SQL statement specified in the SQL property of the TDBISAMQuery component is a SELECT statement, the query is executed using the new field values each time the record pointer in the other data source is changed. It is not necessary to call the Open method of the TDBISAMQuery component 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 data source.

Note
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 manually set, either by using the Params property or the ParamByName method, will be overridden with automatic values.

TDBISAMQuery.EngineVersion Property

```
__property System::UnicodeString EngineVersion
```

Indicates the current version of DBISAM being used. This property is read-only, but published so that it is visible in the Object Inspector in Delphi, Kylix, and C++Builder.

TDBISAMQuery.ExecutionTime Property

```
__property double ExecutionTime
```

The ExecutionTime property indicates the total time, in seconds, that the current SQL statement or script took to execute. This time does not include any time taken to prepare and parse the query, only the execution time itself. If executing multiple SQL statements separated by semicolons (a script), this property will reflect the cumulative execution time of all of the SQL statements.

TDBISAMQuery.GeneratePlan Property

```
__property bool GeneratePlan
```

The GeneratePlan property can be used to specify that a query plan be generated and stored in the Plan property when the SQL statement(s) specified in the SQL property is/are executed.

Note
Query plans are only generated for SQL SELECT, INSERT, UPDATE, or DELETE statements.

TDBISAMQuery.LocaleID Property

```
__property int LocaleID
```

The LocaleID property indicates the locale ID of the result set for the SQL SELECT statement currently specified in the SQL property. This property is only populated when the current SQL statement is a SELECT statement.

Note
This property is only valid after the current SQL SELECT statement has been prepared by calling the Prepare method, or by executing the SQL statement using the Open or ExecSQL methods.

TDBISAMQuery.MaxRowCount Property

```
__property int MaxRowCount
```

Use the MaxRowCount property to control the maximum number of rows that will be returned when executing an SQL SELECT statement. Setting this property to -1 will indicate that the number of rows returned is unlimited.

Note
This property does not respect any DISTINCT, GROUP BY, or ORDER BY claues in the SQL statement. It is primarily useful for making sure that end users do not accidentally construct SQL queries that generate cartesian products or other types of queries that can cause the number of rows to be returned to be enormous.

TDBISAMQuery.ParamCheck Property

```
__property bool ParamCheck
```

Use the ParamCheck property to specify whether or not the Params property is cleared and regenerated if an application modifies the SQL property at runtime. By default the ParamCheck property 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.

Note
The TDBISAMQuery component always behaves like the ParamCheck property is set to True at design-time. The ParamCheck property setting is only respected at runtime.

TDBISAMQuery.ParamCount Property

```
__property System::Word ParamCount
```

Use the ParamCount property to determine how many parameters are in the Params property. If the Params property is True, the ParamCount property always corresponds to the number of actual parameters in the SQL statement specified in the SQL property.

Note
An application can add or delete parameters to the Params property. Such additions and deletions are automatically reflected in ParamCount.

TDBISAMQuery.Params Property

```
__property TDBISAMParams* Params
```

Use the Params property to specify the parameters for an SQL statement. The Params proerty is a zero-based array of TDBISAMParam parameter objects. Index specifies the array element to access.

Note
An easier way to set and retrieve parameter values when the name of each parameter is known is to call the ParamByName method.

TDBISAMQuery.Plan Property

```
__property System::Classes::TStrings* Plan
```

The Plan property is where the query plan is stored when the SQL statement(s) specified in the SQL property is/are executed and the GeneratePlan property is set to True. The Plan property is cleared before each new SQL statement specified in the SQL property is executed.

Note
Query plans are only generated for SQL SELECT, INSERT, UPDATE, or DELETE statements.

TDBISAMQuery.Prepared Property

```
__property bool Prepared
```

Use the Prepared property to determine if an SQL statement is already prepared for execution. If Prepared is True, the SQL statement is prepared, and if Prepared is False, the SQL statement is not prepared. While an SQL statement need not be prepared before execution, execution performance is enhanced if the SQL statement is prepared beforehand, particularly if it is a parameterized SQL statement that is executed more than once using the same parameter values.

Note
An application can change the current setting of Prepared to prepare or unprepare an SQL statement. If Prepared is True, setting it to False calls the UnPrepare method to unprepare the SQL statement. If Prepared is False, setting it to True calls the Prepare method to prepare the SQL statement.

TDBISAMQuery.ReadOnly Property

```
__property bool ReadOnly
```

Use the ReadOnly property to specify that the contents of the query result set generated by a SELECT SQL statement cannot be edited by the application. By default, DBISAM allows all result sets, either live or canned, to be edited.

TDBISAMQuery.RequestLive Property

```
__property bool RequestLive
```

Use the RequestLive property to specify whether or not DBISAM should attempt to return a live result set when the current SELECT SQL statement is excuted. The RequestLive property is False by default, meaning that a canned result set will be returned. Set the RequestLive property to True to request a live result set.

Note
Setting RequestLive to True does not guarantee that a live result set is returned by DBISAM. A live result set will be returned only if the SELECT SQL statement syntax conforms to the syntax requirements for a live result set. If the RequestLive property is True, but the syntax does not conform to the requirements, DBISAM returns a canned result set. After executing the query, inspect the ResultIsLive property to determine whether the request for a live result set was successful.

TDBISAMQuery.ResultIsLive Property

```
__property bool ResultIsLive
```

The ResultIsLive property indicates whether the current SELECT SQL statement returned a live result set.

TDBISAMQuery.RowsAffected Property

```
__property int RowsAffected
```

Use the RowsAffected property to determine how many rows were inserted, updated or deleted by the execution of the current SQL statement specified via the SQL property. If RowsAffected is 0, the SQL statement did not insert, update or delete any rows.

Note
This property is only useful for INSERT, UPDATE, or DELETE SQL statements and will always be equal to the RecordCount property for any SELECT SQL statement that returns a result set.

TDBISAMQuery.SQL Property

```
__property System::Classes::TStrings* SQL
```

Use the SQL property to specify the text of the SQL statement that the TDBISAMQuery component executes when its Open or ExecSQL methods are called. The SQL property may contain multiple SQL statements as long as they are separated by semicolons (;).

TDBISAMQuery.SQLStatementType Property

```
__property TSQLStatementType SQLStatementType
```

The SQLStatementType property indicates the kind of SQL statement currently specified in the SQL property.

Note
This property is only valid after the current SQL statement has been prepared by calling the Prepare method, or by executing the SQL statement using the Open or ExecSQL methods. Also, the SQLStatementType property is useful when executing multiple SQL statements in a script since it always indicates the type of the current SQL statement being executed.

TDBISAMQuery.StmtHandle Property

```
__property Dbisamsq::TDBISAMStatementManager* StmtHandle
```

The StmtHandle property is for internal use only and is not useful to the application developer using DBISAM.

TDBISAMQuery.TableName Property

```
__property System::UnicodeString TableName
```

The TableName property indicates the target table of the SQL statement currently specified in the SQL property. This property is only populated when the current SQL statement is not a SELECT statement.

Note
This property is only valid after the current SQL statement has been prepared by calling the Prepare method, or by executing the SQL statement using the Open or ExecSQL methods. Also, the TableName property is useful when executing multiple SQL statements in a script since it always indicates the target table of the current SQL statement being executed.

TDBISAMQuery.Text Property

```
__property System::UnicodeString Text
```

The Text property indicates the actual text of the SQL statement passed to DBISAM. For parameterized SQL statements, the Text property contains the SQL statement with parameters replaced by the parameter substitution symbol (?) in place of actual parameter values.

Note
The Text property is useful when executing multiple SQL statements in a script since it always indicates the text of the current SQL statement being executed, not the entire script.

TDBISAMQuery.DefineProperties Method

```
virtual void __fastcall
      DefineProperties(System::Classes::TFiler* Filer)
```

Do not use this method. It is used internally by the TDBISAMQuery component to instantiate published design-time property information.

TDBISAMQuery.ExecSQL Method

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

Call the ExecSQL method to execute the SQL statement currently assigned to the SQL property. Use the ExecSQL method to execute any type of SQL statement, including scripts comprised of multiple SQL statements. If the SQL statement is a SELECT SQL statement or it ends with a SELECT SQL statement (such as with scripts), then the ExecSQL method will automatically call the Open method to open the query result set returned by the SELECT statement.

The ExecSQL method prepares the SQL statement or statements in the SQL property for execution if they have not already been prepared. To speed performance in situations where an SQL statement will be executed multiple times with parameters, an application should ordinarily call the Prepare method before calling the ExecSQL method for the first time.

TDBISAMQuery.GetDetailLinkFields Method

```
virtual void __fastcall
      GetDetailLinkFields(System::Classes::TList* MasterFields,
      System::Classes::TList* DetailFields)
```

Call the GetDetailLinkFields method to fill two lists of TField objects that define a master-detail relationship between the query result set and another master datasource. The MasterFields parameter is filled with fields from the master data source whose values must equal the values of the fields in the DetailFields parameter. The DetailFields parameter is filled with fields from the query result set.

TDBISAMQuery.ParamByName Method

```
TDBISAMParam* __fastcall ParamByName(const System::UnicodeString
      Value)
```

Call the ParamByName method to set or access parameter information for a specific parameter based on its name. Value is the name of the parameter to access.

TDBISAMQuery.Prepare Method

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

Call the Prepare method to have DBISAM allocate resources for the execution of an SQL statement, parse the SQL statement, and perform the process of setting up the SQL statement for execution by opening up source tables, etc. The SQL statement is specified via the SQL property.

DBISAM automatically prepares an SQL statement if it is executed without first being prepared. After execution, DBISAM unprepares the SQL statement. When an SQL statement will be executed a number of times, an application should always explicitly prepare the SQL statement using the Prepare method to avoid multiple and unnecessary prepares and unprepares.

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

Note
When you change the SQL property, the current SQL statement is automatically closed and unprepared.

TDBISAMQuery.SaveToTable Method

```
void __fastcall SaveToTable(const System::UnicodeString
      NewDatabaseName, const System::UnicodeString NewTableName)
```

Call the SaveToTable method to save the contents of a query result set to a permanent table for use in subsequent database operations or queries.

Note
If you wish to store the result set in a table in a different database you must provide a different database directory, for local sessions, or database name, for remote sessions, in the NewDatabaseName parameter.

TDBISAMQuery.TDBISAMQuery Method

```
__fastcall virtual TDBISAMQuery(System::Classes::TComponent*
      AOwner)
```

Use the New operator to create an instance of a TDBISAMQuery component using the constructor.

TDBISAMQuery.UnPrepare Method

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

Call the UnPrepare method to free the resources allocated for an SQL statement previously prepared with the Prepare method.

TDBISAMQuery.AfterExecute Event

```
__property System::Classes::TNotifyEvent AfterExecute
```

The AfterExecute event is fired after the execution of any SQL statement using the ExecSQL or Open method. If executing multiple SQL statements in a script, the AfterExecute event will fire after each SQL statement in the script.

TDBISAMQuery.BeforeExecute Event

```
__property System::Classes::TNotifyEvent BeforeExecute
```

The BeforeExecute event is fired before the execution of any SQL statement using the ExecSQL or Open method. If executing multiple SQL statements in a script, the BeforeExecute event will fire before each SQL statement in the script.

TDBISAMQuery.OnAlterProgress Event

```
__property TProgressEvent OnAlterProgress
```

The OnAlterProgress event is fired when the structure of a table is altered by executing an ALTER TABLE SQL statement using the ExecSQL or Open method. Use the PercentDone parameter to display progress information in your application while the table's structure is being altered.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMQuery.OnCopyProgress Event

```
__property TProgressEvent OnCopyProgress
```

The OnCopyProgress event is fired when a table is copied to a new table name by executing a COPY TABLE SQL statement using the ExecSQL or Open method. Use the PercentDone parameter to display progress information in your application while the table is copied.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMQuery.OnDataLost Event

```
__property TDataLostEvent OnDataLost
```

The OnDataLost event is fired when executing an ALTER TABLE or CREATE INDEX StatementSQL statement using the ExecSQL or Open method and a change in the structure of the table has caused data to be lost or the addition of a unique index has caused a key violation.

The Cause parameter allows you to determine the cause of the data loss.

The ContextInfo parameter allows you to determine the exact field, index, or table name that is causing or involved in the loss of data.

The Continue parameter allows you to abort the table structure alteration of index addition process and return the table to it's original state with all of the data intact.

The StopAsking parameter allows you to tell DBISAM to stop reporting data loss problems and simply complete the operation.

Note
You may set the Continue parameter to True several times and at a later time set the Continue parameter to False and still have the table retain its original content and structure.

TDBISAMQuery.OnExportProgress Event

```
__property TProgressEvent OnExportProgress
```

The OnExportProgress event is fired when a table is exported to a text file by executing an EXPORT TABLE SQL statement using the ExecSQL or Open method. Use the PercentDone parameter to display progress information in your application while the table is exported.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMQuery.OnGetParams Event

```
__property System::Classes::TNotifyEvent OnGetParams
```

The OnGetParams event is fired before the execution of any SQL statement using the ExecSQL or Open method, but after the SQL statement is prepared. This event gives the application an opportunity to dynamically populate parameters in an SQL statement prior to the SQL statement being executed. This event is especially useful for scripts that contain multiple parameterized SQL statements.

TDBISAMQuery.OnImportProgress Event

```
__property TProgressEvent OnImportProgress
```

The OnImportProgress event is fired when a table is imported from a text file by executing an IMPORT TABLE SQL statement using the ExecSQL or Open method. Use the PercentDone parameter to display progress information in your application while the table is imported.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMQuery.OnIndexProgress Event

```
__property TProgressEvent OnIndexProgress
```

The OnIndexProgress event is fired when a new index is added to a table by executing a CREATE INDEX SQL statement using the ExecSQL or Open method. Use the PercentDone parameter to display progress information in your application while the index is being added.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMQuery.OnLoadFromStreamProgress Event

```
__property TProgressEvent OnLoadFromStreamProgress
```

The OnLoadFromStreamProgress event is fired when a stream is loaded into a query result set using the LoadFromStream method. Use the PercentDone parameter to display progress information in your application while the query result set is being loaded from the stream.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMQuery.OnOptimizeProgress Event

```
__property TProgressEvent OnOptimizeProgress
```

The OnOptimizeProgress event is fired when a table is optimized by executing an OPTIMIZE TABLE SQL statement using the ExecSQL or Open method. Use the PercentDone parameter to display progress information in your application while the table is being optimized.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMQuery.OnQueryError Event

```
__property TAbortErrorEvent OnQueryError
```

The OnQueryError event is fired whenever there is an error of any kind encountered in the preparation or execution of an SQL statement. This event is especially useful for scripts that contain multiple SQL statements since this event offers the ability to continue with a script even though one of the SQL statements has encountered an error.

TDBISAMQuery.OnQueryProgress Event

```
__property TAbortProgressEvent OnQueryProgress
```

The OnQueryProgress event is fired when an SQL statement is executed using the ExecSQL or Open method. Use the PercentDone parameters to display progress information in your application while the SQL statement is executing. Setting the Abort parameter to True at any time during the execution of the SQL statement will cause the execution to stop. This event is not triggered for live query result sets. Please see the Live Queries and Canned Queries topic for more information.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMQuery.OnRepairLog Event

```
__property TLogEvent OnRepairLog
```

The OnRepairLog event is fired when a table is repaired by executing a REPAIR TABLE SQL statement using the ExecSQL or Open method and DBISAM needs to indicate the current status of the repair (such as the start or finish) or an error is found in the integrity of the table. Use the LogMesssage parameter to display repair log information in your application while the table is being repaired or to save the log messages to a file for later viewing.

TDBISAMQuery.OnRepairProgress Event

```
__property TSteppedProgressEvent OnRepairProgress
```

The OnRepairProgress event is fired when a table is repaired by executing a REPAIR TABLE SQL statement using the ExecSQL or Open method. Use the Step and PercentDone parameters to display progress information in your application while the table is being repaired.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMQuery.OnSaveProgress Event

```
__property TProgressEvent OnSaveProgress
```

The OnSaveProgress event is fired when a query result set saved to a table using the SaveToTable method. Use the PercentDone parameter to display progress information in your application while the result set is being saved to the table.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMQuery.OnSaveToStreamProgress Event

```
__property TProgressEvent OnSaveToStreamProgress
```

The OnSaveToStreamProgress event is fired when a query result set is saved to a stream using the SaveToStream method. Use the PercentDone parameter to display progress information in your application while the query result set is being saved to the stream.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMQuery.OnSQLChanged Event

```
__property System::Classes::TNotifyEvent OnSQLChanged
```

The OnSQLChanged event is fired whenever the SQL property is modified.

TDBISAMQuery.OnUpgradeLog Event

```
__property TLogEvent OnUpgradeLog
```

The UpgradeLog event is fired when a table is upgraded from an old table format by executing an UPGRADE TABLE SQL statement using the ExecSQL or Open method. Use the LogMesssage parameter to display upgrade log information in your application while the table is being upgraded or to save the log messages to a file for later viewing.

TDBISAMQuery.OnUpgradeProgress Event

```
__property TSteppedProgressEvent OnUpgradeProgress
```

The OnUpgradeProgress event is fired when a table is upgraded from an old table format by executing an UPGRADE TABLE SQL statement using the ExecSQL or Open method. Use the Step and PercentDone parameters to display progress information in your application while the table is being upgraded.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMQuery.OnVerifyLog Event

```
__property TLogEvent OnVerifyLog
```

The OnVerifyLog event is fired when a table is verified by executing a VERIFY TABLE SQL statement using the ExecSQL or Open method and DBISAM needs to indicate the current status of the verification (such as the start or finish) or an error is found in the integrity of the table. Use the LogMesssage parameter to display verification log information in your application while the table is being verified or to save the log messages to a file for later viewing.

TDBISAMQuery.OnVerifyProgress Event

```
__property TSteppedProgressEvent OnVerifyProgress
```

Occurs when a table is verified by executing a VERIFY TABLE SQL statement using the ExecSQL or Open method. Use the Step and PercentDone parameters to display progress information in your application while the table is being repaired.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

5.20 TDBISAMRecord Component

Header File: dbisamtb

Inherits From TObject

Use the TDBISAMRecord object to access and/or update the current record in the scope of a before and/or after trigger, or error event in the TDBISAMEngine component. Please see the Customizing the Engine topic for more information.

Properties	Methods	Events
FieldCount	CreateBlobStream	
Fields	FieldByName	
FieldValues	FindField	
Modified	GetBlobFieldData	
RecNo	GetFieldNames	
RecordHash	TDBISAMRecord	
RecordID		
RecordSize		

TDBISAMRecord.FieldCount Property

```
__property int FieldCount
```

The FieldCount property indicates the number of fields present in the current record. You can use the FieldCount property to iterate through the 0-based Fields property and access fields by their ordinal position in the current record.

TDBISAMRecord.Fields Property

```
__property Data::Db::TFields* Fields
```

The Fields property allows you to access a given TField in the current record by its ordinal position (0-based) in the current record.

TDBISAMRecord.FieldValues Property

```
__property System::Variant FieldValues[const System::UnicodeString FieldName]
```

The FieldValues property allows you read or update the value of a given field using its field name.

Note
The FieldValues property only returns field values as variants, so the usual caveats regarding variants and NULL values applies.

TDBISAMRecord.Modified Property

```
__property bool Modified
```

The Modified property indicates whether the current record has been modified since the insert, update, or delete operation started. The Modified property is initially False at the beginning of an insert, update or delete, and is only set to True if a Before or After trigger, or an error handler, modifies the current record.

TDBISAMRecord.RecNo Property

```
__property int RecNo
```

The RecNo property indicates the physical record number of the current record.

Note
This value is not necessarily the same as the RecNo property for the TDBISAMTable or TDBISAMQuery components, which is a logical record number.

TDBISAMRecord.RecordHash Property

```
__property System::UnicodeString RecordHash
```

The RecordHash property indicates the MD5 hash value of the current record in the form of a string.

TDBISAMRecord.RecordID Property

```
__property int RecordID
```

The RecordID property indicates the inviolate record ID of the current record.

TDBISAMRecord.RecordSize Property

```
__property System::Word RecordSize
```

The RecordSize property indicates the total physical size, in bytes, of the current record.

TDBISAMRecord.CreateBlobStream Method

```
System::Classes::TStream* __fastcall
      CreateBlobStream(Data::Db::TField* Field,
      Data::Db::TBlobStreamMode Mode)
```

Call the CreateBlobStream method to obtain a stream for reading data from or writing data to a BLOB field. The Field parameter must specify a TBlobField component from the Fields property. The Mode parameter specifies whether the stream will be used for reading, writing, or updating the contents of the field.

TDBISAMRecord.FieldByName Method

```
Data::Db::TField* __fastcall FieldByName(const
      System::UnicodeString FieldName)
```

Call the FieldByName method to access a TField component by its name. If the specified field does not exist, an EDatabaseError exception is triggered.

TDBISAMRecord.FindField Method

```
Data::Db::TField* __fastcall FindField(const
      System::UnicodeString FieldName)
```

Call the FindField method to determine if a specified TField component exists in the current record. If the FindField method finds a field with a matching name, it returns the TField component for the specified field. Otherwise it returns nil.

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

TDBISAMRecord.GetBlobFieldData Method

```
int __fastcall GetBlobFieldData(int FieldNo,
      System::DynamicArray<System::Byte> &Buffer)
```

Call the GetBlobFieldData method to read BLOB data from the field specified by FieldNo direcctory into a buffer. The buffer is a dynamic array of bytes, so that it can grow to accommodate the size of the BLOB data. The GetBlobFieldData method returns the size of the buffer.

TDBISAMRecord.GetFieldNames Method

```
void __fastcall GetFieldNames(System::Classes::TStrings* List)
```

Call the GetFieldNames method to retrieve the list of fields for the current record into a TStrings object.

Note
The TStrings object must be created by the application prior to calling this method, and the application is responsible for its destruction after the method is called.

TDBISAMRecord.TDBISAMRecord Method

```
__fastcall TDBISAMRecord(void)
```

Use the New operator to create an instance of a TDBISAMRecord object using the constructor. However, you should never need to create an instance of the TDBISAMRecord object since one is automatically created when any of the TDBISAMEngine trigger or error events is called.

5.21 TDBISAMSession Component

Header File: dbisamtb

Inherits From Classes

Use the TDBISAMSession component to manage a local or remote session within an application. A session acts like a "virtual user" and each new session component used in an application maintains its own database connections, table buffers, table cursors, etc. Because of the unique requirements of a multi-threaded application, DBISAM requires that you use a separate TDBISAMSession component for each thread in use, thus treating each thread as a separate "virtual user".

A default TDBISAMSession component is created automatically when the application is started and can be referenced via the global Session function in the dbisamtb unit (Delphi) and dbisamtb header file (C++).

Note
Applications that maintain multiple sessions can manage them through the TDBISAMEngine component. A TDBISAMEngine component is created automatically when an application is started and can be referenced via the global Engine function in the dbisamtb unit (Delphi) and dbisamtb header file (C++).

Properties	Methods	Events
Active	AddPassword	OnPassword
AutoSessionName	AddRemoteDatabase	OnRemoteLogin
CurrentRemoteID	AddRemoteDatabaseUser	OnRemoteProcedureProgress
CurrentRemoteUser	AddRemoteEvent	OnRemoteReceiveProgress
CurrentServerUser	AddRemoteProcedure	OnRemoteReconnect
CurrentServerUserAddress	AddRemoteProcedureUser	OnRemoteSendProgress
DatabaseCount	AddRemoteUser	OnRemoteTimeout
Databases	CallRemoteProcedure	OnRemoteTrace
EngineVersion	Close	OnShutdown
ForceBufferFlush	CloseDatabase	OnStartup
Handle	DeleteRemoteDatabase	
KeepConnections	DeleteRemoteDatabaseUser	
LockProtocol	DeleteRemoteEvent	
LockRetryCount	DeleteRemoteProcedure	
LockWaitTime	DeleteRemoteProcedureUser	
PrivateDir	DeleteRemoteUser	
ProgressSteps	DisconnectRemoteSession	
RemoteAddress	DropConnections	

RemoteCompression	FindDatabase	
RemoteEncryption	GetDatabaseNames	
RemoteEncryptionPassword	GetPassword	
RemoteHost	GetRemoteAdminAddress	
RemoteParams	GetRemoteAdminPort	
RemotePassword	GetRemoteAdminThreadCacheSize	
RemotePing	GetRemoteConfig	
RemotePingInterval	GetRemoteConnectedSessionCount	
RemotePort	GetRemoteDatabase	
RemoteService	GetRemoteDatabaseNames	
RemoteTimeout	GetRemoteDatabaseUser	
RemoteTrace	GetRemoteDatabaseUserNames	
RemoteUser	GetRemoteDateTime	
SessionName	GetRemoteEngineVersion	
SessionType	GetRemoteEvent	
StoreActive	GetRemoteEventNames	
StrictChangeDetection	GetRemoteLogCount	
	GetRemoteLogRecord	
	GetRemoteMainAddress	
	GetRemoteMainPort	
	GetRemoteMainThreadCacheSize	
	GetRemoteMemoryUsage	
	GetRemoteProcedure	
	GetRemoteProcedureNames	
	GetRemoteProcedureUser	
	GetRemoteProcedureUserNames	
	GetRemoteServerDescription	
	GetRemoteServerName	
	GetRemoteSessionCount	
	GetRemoteSessionInfo	
	GetRemoteUpTime	
	GetRemoteUser	
	GetRemoteUserNames	
	GetRemoteUTCDateTime	
	GetTableNames	
	ModifyRemoteConfig	

	ModifyRemoteDatabase	
	ModifyRemoteDatabaseUser	
	ModifyRemoteEvent	
	ModifyRemoteProcedure	
	ModifyRemoteProcedureUser	
	ModifyRemoteUser	
	ModifyRemoteUserPassword	
	Open	
	OpenDatabase	
	RemoteParamByName	
	RemoveAllPasswords	
	RemoveAllRemoteMemoryTables	
	RemovePassword	
	RemoveRemoteSession	
	SendProcedureProgress	
	StartRemoteServer	
	StopRemoteServer	
	TDBISAMSession	

TDBISAMSession.Active Property

```
__property bool Active
```

Use the Active property to specify whether or not a session is active. Setting Active to True starts the session and triggers the OnStartup event for the session. If the SessionType property is set to stRemote, then DBISAM will attempt to connect to the database server specified by the RemoteHost or RemoteAddressand RemotePort or RemoteService properties. If the session can successfully connect to the database server, it will then automatically login to the server using the RemoteUser and RemotePasword properties.

Setting Active to False closes any open datasets, and disconnects active database connections. If the SessionType property is set to stRemote, then the connection to the database server is closed and the user is logged out.

TDBISAMSession.AutoSessionName Property

```
__property bool AutoSessionName
```

Use the AutoSessionName property to specify whether or not a unique session name is automatically generated for the TDBISAMSession component. AutoSessionName is intended to guarantee developers of multi-threaded applications that TDBISAMSession components created for each thread are assigned unique names at runtime.

When AutoSessionName is False (the default), the application must set the SessionName property for a session component to a unique name within the context of the application. When AutoSessionName is True, the TDBISAMSession component assigns the SessionName property automatically and replicates this session name across the SessionName properties of all TDBISAMDatabase, TDBISAMQuery, and TDBISAMTable components in the data module or form where the session component is created. This allows applications to use TDBISAMSession components in data modules that are replicated over multiple threads without having to worry about providing unique names for each session when the data module is created. The TDBISAMSession component constructs a session name by taking the current value of the Name property and appending an underscore (_) followed by a numeric value. For example, if the Name property was set to "CustomerSession", then the AutoSessionName property would be set to "CustomerSession_2" for the second session created.

Note
The following restrictions apply to the AutoSessionName property:

- You cannot set the AutoSessionName property to True for a TDBISAMSession component in a data module or form that contains more than one TDBISAMSession component.

- You cannot add a TDBISAMSession component to a data module or form that already contains a TDBISAMSession component with its AutoSessionName property set to True.

- You cannot directly set the SessionName property of a TDBISAMSession component when its AutoSessionName property is True.

TDBISAMSession.CurrentRemoteID Property

```
__property int CurrentRemoteID
```

Indicates the ID of the session that is currently logged in to a database server.

TDBISAMSession.CurrentRemoteUser Property

```
__property System::UnicodeString CurrentRemoteUser
```

Indicates the user name of the session that is currently logged in to a database server.

Note
This property is only valid when the current session is a remote session (SessionType=stRemote) and is successfully logged into a DBISAM database server.

TDBISAMSession.CurrentServerUser Property

```
__property System::UnicodeString CurrentServerUser
```

Indicates the user name of the session that is currently logged in to a database server.

Note
This property is only used from within triggers or server-side procedures when the engine is behaving as a database server and the TDBISAMEngine EngineType is equal to etServer. It returns an empty string at any other time.

TDBISAMSession.CurrentServerUserAddress Property

```
__property System::UnicodeString CurrentServerUserAddress
```

Indicates the IP address of the session that is currently logged in to a database server.

Note
This property is only used from within triggers or server-side procedures when the engine is behaving as a database server and the TDBISAMEngine EngineType is equal to etServer. It returns an empty string at any other time.

TDBISAMSession.DatabaseCount Property

```
__property int DatabaseCount
```

Indicates the number of active TDBISAMDatabase components currently associated with the session. This number can change as TDBISAMDatabase components are opened and closed. If the DatabaseCount property is zero, there are currently no active TDBISAMDatabase components associated with the session.

DatabaseCount is typically used with the Databases property to iterate through the current set of active TDBISAMDatabase components in a session.

TDBISAMSession.Databases Property

```
__property TDBISAMDatabase* Databases[int Index]
```

Use the Databases property to access active TDBISAMDatabase components associated with a session. An active TDBISAMDatabase component is one that has its Connected property set to True.

The Databases property is typically used with the DatabaseCount property to iterate through the current set of active TDBISAMDatabase components in a session.

TDBISAMSession.EngineVersion Property

```
__property System::UnicodeString EngineVersion
```

Indicates the current version of DBISAM being used. This property is read-only, but published so that it is visible in the Object Inspector in Delphi, Kylix, and C++Builder.

TDBISAMSession.ForceBufferFlush Property

```
__property bool ForceBufferFlush
```

Use the ForceBufferFlush property to specify that the all TDBISAMQuery and TDBISAMTable components in this session should automatically force the operating system to flush its write buffers to disk after DBISAM has written any data using operating system calls. This can significantly reduce instances of corruption in the event of an improper application shutdown, however it can also cause performance degradation for batch updates, repairing tables, etc. A better alternative for reducing the performance implications of this property is to use the FlushBuffers method of the TDBISAMTable or TDBISAMQuery components to selectively flush the operating system write buffers to disk as necessary.

TDBISAMSession.Handle Property

```
__property Dbisamen::TDBISAMSessionManager* Handle
```

The Handle property is for internal use only and is not useful to the application developer using DBISAM.

TDBISAMSession.KeepConnections Property

```
__property bool KeepConnections
```

Use the KeepConnections property to specify whether or not a temporary TDBISAMDatabase component created in the context of a session maintains a database connection even if there are no active TDBISAMQuery or TDBISAMTable components associated with the TDBISAMDatabase component. If the KeepConnections property is True (the default), the application maintains TDBISAMDatabase connections until the application exits or calls the DropConnections method. For remote sessions, the KeepConnections property should remain True to reduce network traffic and avoid constantly opening and closing databases.

When the KeepConnections property is False, an application disconnects from a database when all TDBISAMQuery and TDBISAMTable components associated with a TDBISAMDatabase component are closed. Dropping a connection releases system resources allocated to the connection, but if a dataset is later reopened that uses the same database, the connection must be reestablished and initialized.

Note
The duration of a connection for a persistent, not temporary, TDBISAMDatabase component is determined by the TDBISAMDatabase component's KeepConnection property instead of the session's KeepConnections property.

TDBISAMSession.LockProtocol Property

```
__property TLockProtocol LockProtocol
```

Use the LockProtocol property to specify whether the session will use a pessimistic or optimistic locking model when editing records via navigational or SQL methods. The pessimistic locking model dictates that records should be locked when the record is retrieved for editing, which is during the Edit method of a TDBISAMTable or TDBISAMQuery component and during the execution of an SQL UPDATE statement. The optimistic locking model dictates that records should be locked when the record modifications are posted to the database table, which is during the Post method of a TDBISAMTable or TDBISAMQuery component and during the execution of an SQL UPDATE statement. Using an optimistic locking model for remote connections to a database server removes the possibility that dangling record locks will be left on the server if a client application is terminated unexpectedly.

The default value is lpPessimistic.

TDBISAMSession.LockRetryCount Property

```
__property System::Byte LockRetryCount
```

Use the LockRetryCount property to specify the number of times DBISAM will retry a record lock or table lock before displaying a lock failure message. The amount of time between each lock retry is controlled by the LockWaitTime property of the TDBISAMSession component.

Note
This property only affects datasets (TDBISAMTable or TDBISAMQuery components) attached to this TDBISAMSession component via their SessionName property.

TDBISAMSession.LockWaitTime Property

```
__property System::Word LockWaitTime
```

Use the LockWaitTime property to specify the amount of time, in milliseconds, DBISAM will wait between retries of a record lock or table lock. The number of times that a lock is retried is controlled by the LockRetryCount property of the TDBISAMSession component.

Note
This property only affects datasets (TDBISAMTable or TDBISAMQuery components) attached to this TDBISAMSession component via their SessionName property.

TDBISAMSession.PrivateDir Property

```
__property System::UnicodeString PrivateDir
```

Use the PrivateDir property to set the physical directory in which to store temporary tables such as those generated by DBISAM to store canned query result sets of SQL SELECT statements. This property defaults to the local temporary files directory for the current user.

Note
This property is ignored for remote sessions. The database server has a specific configuration setting for the location of temporary tables that are created on the database server.

TDBISAMSession.ProgressSteps Property

```
__property System::Word ProgressSteps
```

Use the ProgressSteps property to specify how many times a progress event will be fired during a batch operation on any TDBISAMQuery or TDBISAMTable component attached to this TDBISAMSsion component via their SessionName property. This property can be set to any value between 0 and 100. Setting this property to 0 will suppress all progress events.

TDBISAMSession.RemoteAddress Property

```
__property System::UnicodeString RemoteAddress
```

Use the RemoteAddress property to specify the IP address of a database server that you wish to connect to. This property only applies to remote sessions where the SessionType property is set to stRemote. When the session is opened via the Open method or by setting the Active property to True, DBISAM will attempt to connect to the database server specified by the RemoteAddress or RemoteHost and RemotePort or RemoteService properties.

TDBISAMSession.RemoteCompression Property

```
__property System::Byte RemoteCompression
```

Use the RemoteCompression property to set the level of compression used for a remote session. This property only applies to remote sessions where the SessionType property is set to stRemote. The compression is specified as a Byte value between 0 and 9, with the default being 0, or none, and 6 being the best selection for size/speed. The default compression is ZLib, but can be replaced by using the TDBISAMEngine events for specifying a different type of compression. Please see the Compression and Customizing the Engine topics for more information.

Note
This property can be changed while the session is connected so that you may adjust the level of compression for individual situations.

TDBISAMSession.RemoteEncryption Property

```
__property bool RemoteEncryption
```

Use the RemoteEncryption property to specify that a remote session will be encrypted using the RemoteEncryptionPassword property. This property only applies to remote sessions where the SessionType property is set to stRemote. The default encryption uses the 128-bit Blowfish algorithm, but can be replaced by using the TDBISAMEngine events for specifying a different type of block-cipher encryption. Please see the Encryption and Customizing the Engine topics for more information.

Note
This property must be set prior to connecting the session to the database server via the Open method or the Active property.

TDBISAMSession.RemoteEncryptionPassword Property

```
__property System::UnicodeString RemoteEncryptionPassword
```

Use the RemoteEncryptionPassword property to specify the password for an encrypted remote session. The RemoteEncryption property controls whether the session is encrypted or not. This property only applies to remote sessions where the SessionType property is set to stRemote. The default encryption uses the 128-bit Blowfish algorithm, but can be replaced by using the TDBISAMEngine events for specifying a different type of block-cipher encryption. Please see the Encryption and Customizing the Engine topics for more information.

Note
This property must be set prior to connecting the session to the database server via the Open method or the Active property.

TDBISAMSession.RemoteHost Property

```
__property System::UnicodeString RemoteHost
```

Use the RemoteHost property to specify the host name of a database server that you wish to connect to. A host name is alternate way of specifying a remote IP address by relying on DNS to translate the host name into a usable IP address. This property only applies to remote sessions where the SessionType property is set to stRemote. When the session is opened via the Open method or by setting the Active property to True, DBISAM will attempt to connect to the database server specified by the RemoteAddress or RemoteHost and RemotePort or RemoteService properties.

TDBISAMSession.RemoteParams Property

```
__property TDBISAMParams* RemoteParams
```

Use the RemoteParams property to manipulate TDBISAMParam objects when calling server-side procedures from a remote session using the CallRemoteProcedure method or from within a server-side procedure via a TDBISAMEngine OnServerProcedure event handler. You should populate the RemoteParams property with the desired parameters prior to calling the CallRemoteProcedure method and, once the server-side procedure returns, you can then inspect the RemoteParams property to retrieve the return values, if any, from the server-side procedure.

TDBISAMSession.RemotePassword Property

```
__property System::UnicodeString RemotePassword
```

Use the RemotePassword property to specify the password for automating the login to a database server. This property only applies to remote sessions where the SessionType property is set to stRemote. When the session is opened via the Open method or by setting the Active property to True, DBISAM will attempt to connect to the database server specified by the RemoteAddress or RemoteHost and RemotePort or RemoteService properties, and automatically login to the server using the RemoteUser and RemotePassword properties. If for any reason these properties are not set correctly then the OnRemoteLogin event will fire. If an event handler is not assigned to the OnRemoteLogin event then a remote login dialog will be displayed in order to prompt the user for a user name and password.

TDBISAMSession.RemotePing Property

```
__property bool RemotePing
```

Use the RemotePing property to enable or disable pinging to a database server. Pinging the database server allows for the use of a smaller dead session expiration time and can be used to prevent dangling locks when a client workstation shuts down and leaves an open session on the database server. When the RemotePing property is set to True, the remote session will ping the database server according to the interval in seconds specified by the RemotePingInterval property. This property only applies to remote sessions where the SessionType property is set to stRemote.

TDBISAMSession.RemotePingInterval Property

```
__property System::Word RemotePingInterval
```

Use the RemotePingInterval property to specify the interval in seconds between pings to a database server when the RemotePing property is set to True. This property only applies to remote sessions where the SessionType property is set to stRemote.

TDBISAMSession.RemotePort Property

```
__property int RemotePort
```

Use the RemotePort property to specify the port of a database server that you wish to connect to. This property only applies to remote sessions where the SessionType property is set to stRemote. When the session is opened via the Open method or by setting the Active property to True, DBISAM will attempt to connect to the database server specified by the RemoteAddress or RemoteHost and RemotePort or RemoteService properties.

Note
A database server listens on two different ports, one for normal data connections and one for administrative connections. Be sure to set the correct port using this property or you will get errors when trying to execute administrative functions on the normal data port or vice-versa. This is especially important since the administrative port requires encrypted connections (RemoteEncryption=True).

TDBISAMSession.RemoteService Property

```
__property System::UnicodeString RemoteService
```

Use the RemoteService property to specify the service name of a database server that you wish to connect to. A service name is an alternate way of specifying a remote port using a standard name instead of a port number. This property only applies to remote sessions where the SessionType property is set to stRemote. When the session is opened via the Open method or by setting the Active property to True, DBISAM will attempt to connect to the database server specified by the RemoteAddress or RemoteHost and RemotePort or RemoteService properties.

Note
A database server listens on two different ports, one for normal data connections and one for administrative connections. Be sure to set the correct service name using this property that translates to the correct port on the database server or you will get errors when trying to execute administrative functions on the normal data port or vice-versa. This is especially important since the administrative port requires encrypted connections (RemoteEncryption=True).

TDBISAMSession.RemoteTimeout Property

```
__property System::Word RemoteTimeout
```

Use the RemoteTimeout property to specify the amount of time, in seconds, that a remote session should wait for a response from a database server before firing the OnRemoteTimeout event. If the OnRemoteTimeout event is assigned an event handler, then the event handler can decide whether to disconnect the session or not. If the OnRemoteTimeout event is not assigned an event handler, then DBISAM will disconnect the session. This property only applies to remote sessions where the SessionType property is set to stRemote.

Note
Just because the session disconnects its side of the connection with the server does not necessarily mean that the server knows the session is disconnected or immediately treats the session as a "dead" session. The server may just simply be executing a very long process and has not sent a progress message in a longer period of time than what is configured for the RemoteTimeout property. Please see the DBISAM Architecture topic for more information on the meaning of "dead" sessions on a database server.

TDBISAMSession.RemoteTrace Property

```
__property bool RemoteTrace
```

Use the RemoteTrace property to enable or disable tracing of all requests sent to and responses received from a database server. When the RemoteTrace property is set to True, the OnRemoteTrace event is fired whenever a request is sent to or a response is received from the database server. This can be useful in debugging performance issues with a database server connection. This property only applies to remote sessions where the SessionType property is set to stRemote.

TDBISAMSession.RemoteUser Property

```
__property System::UnicodeString RemoteUser
```

Use the RemoteUser property to specify the user name for automating the login to a database server. This property only applies to remote sessions where the SessionType property is set to stRemote. When the session is opened via the Open method or by setting the Active property to True, DBISAM will attempt to connect to the database server specified by the RemoteAddress or RemoteHost and RemotePort or RemoteService properties, and automatically login to the server using the RemoteUser and RemotePassword properties. If for any reason these properties are not set correctly then the OnRemoteLogin event will fire. If an event handler is not assigned to the OnRemoteLogin event then a remote login dialog will be displayed in order to prompt the user for a user name and password.

TDBISAMSession.SessionName Property

```
__property System::UnicodeString SessionName
```

Use the SessionName property to specify a unique session name that can be used by TDBISAMDatabase, TDBISAMQuery, and TDBISAMTable components to link to this session via their own SessionName properties, which must either match the SessionName property of an active session or be blank, indicating that they should be associated with the default global TDBISAMSession component that is created automatically when the application is started and can be referenced via the global Session function in the dbisamtb unit (Delphi and Kylix) and dbisamtb header file (C++Builder).

Note
If the AutoSessionName property is True, an application cannot set the SessionName property directly.

TDBISAMSession.SessionType Property

```
__property TSessionType SessionType
```

Use the SessionType property to specify the type of session represented by the session component. Setting this property to stLocal (the default) will cause DBISAM to access all databases and tables in the session directly using operating system calls. Setting this property to stRemote will cause DBISAM to access all databases and tables in the session remotely through the database server specified by the RemoteAddress or RemoteHost and RemotePort or RemoteService properties.

Note
This property must be set prior to starting the session via the Open method or the Active property.

TDBISAMSession.StoreActive Property

```
__property bool StoreActive
```

Use the StoreActive property to determine if the session should store the current value of its Active property, and subsequently, the Active/Connected property values of all other DBISAM components such as the TDBISAMDatabase, TDBISAMTable, and TDBISAMQuery components, in the owner form or data module. The default value for this property is True.

TDBISAMSession.StrictChangeDetection Property

```
__property bool StrictChangeDetection
```

Use the StrictChangeDetection property to indicate that all TDBISAMQuery and TDBISAMTable components linked to this TDBISAMSession component will use either a strict or lazy change detection policy when checking for changes made by other users or sessions. The default value is False, meaning that lazy change detection is the default change detection policy.

TDBISAMSession.AddPassword Method

```
void __fastcall AddPassword(const System::UnicodeString
      Password)
```

Call the AddPasssword method to add a password to the session for use when accessing encrypted database tables. If an application opens a table using a TDBISAMQuery or TDBISAMTable component that is encrypted with a password and the session does not currently contain the password in memory, then the session will first try to trigger the OnPassword event. If the OnPassword event does have an event handler assigned to it, then DBISAM will display a password dialog prompting the user for a valid password before allowing access to the encrypted table.

Note
If an application defines an OnPassword event handler, the handler should call the AddPassword method to add passwords for the session.

TDBISAMSession.AddRemoteDatabase Method

```
void __fastcall AddRemoteDatabase(const System::UnicodeString
      DatabaseName, const System::UnicodeString DatabaseDescription,
      const System::UnicodeString ServerPath)
```

Call the AddRemoteDatabase method to add a new database to a database server. Use the DatabaseName parameter to specify the new database name, the DatabaseDescription parameter to give it a description, and the ServerPath parameter to specify the physical path to the tables, relative to the database server.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.AddRemoteDatabaseUser Method

```
void __fastcall AddRemoteDatabaseUser(const
      System::UnicodeString DatabaseName, const System::UnicodeString
      AuthorizedUser, TDatabaseRights RightsToAssign)
```

Call the AddRemoteDatabaseUser method to add rights for an existing user to an existing database on a database server. Use the DatabaseName parameter to specify the existing database name, the AuthorizedUser parameter to specify the existing user, and the RightsToAssign parameter to specify the rights to give to the user for the database. You may use a wildcard (*) for the AuthorizedUser parameter. For example, you could specify just "*" for all users or "Accounting*" for all users whose user name begins with "Accounting".

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.AddRemoteEvent Method

```
void __fastcall AddRemoteEvent(const System::UnicodeString
      EventName, const System::UnicodeString EventDescription,
      TEventRunType EventRunType, System::TDateTime EventStartDate,
      System::TDateTime EventEndDate, System::TDateTime EventStartTime,
       System::TDateTime EventEndTime, System::Word EventInterval,
      TEventDays EventDays, TEventDayOfMonth EventDayOfMonth,
      TEventDayOfWeek EventDayOfWeek, const TEventMonths &EventMonths)
```

Call the AddRemoteEvent method to add a new scheduled event to a database server. Use the EventName parameter to specify the new event name, the EventDescription parameter to give it a description, the EventRunType parameter to specify how the event should be run, the EventStartDate and EventEndDate parameter to specify the dates during which the event should be run, the EventStartTime and EventEndTime parameters to specify the time of day during which the event can be run, the EventInterval to specify how often the event should be run (actual interval unit depends upon the EventRunType, and the EventDays, EventDayOfMonth, EventDayOfWeek, and EventMonths parameters to specify on what day of the week or month the event should be run. The following describes which parameters are required for each possible EventRunType value (all run types require the EventStartDate, EventEndDate, EventStartTime, and EventEndTime parameters):

Run Type	Parameters Needed
rtOnce	No Other Parameters
rtHourly	EventInterval (Hours)
rtDaily	EventInterval (Days)
rtWeekly	EventInterval (Weeks) EventDays **Note** The EventDays parameter specifies which days of the week to run on, with day 1 being Sunday and day 7 being Saturday. Just set the array index of the desired day to True to cause the event to run on that day.
rtMonthly	EventDayOfMonth EventDayOfWeek EventMonths

	Note The EventDayOfMonth parameter specifies which day of the month to run on, a numbered day (1-31) or a specific day (Sunday-Saturday) of the 1st, 2nd, 3rd, or 4th week specified by the EventDayOfWeekParameter. The EventMonths parameter specifies which months of the year to run on, with month 1 being January and month 12 being December. Just set the array index of the desired month to True to cause the event to run on that month.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.AddRemoteProcedure Method

```
void __fastcall AddRemoteProcedure(const System::UnicodeString
      ProcedureName, const System::UnicodeString ProcedureDescription)
```

Call the AddRemoteProcedure method to add a new server-side procedure to a database server. Use the ProcedureName parameter to specify the new procedure name and the ProcedureDescription parameter to give it a description. This method only identifies the procedure to the database server for the purposes of allowing user rights to be assigned to the server-side procedure. The actual server-side procedure itself must be implemented via a TDBISAMEngine OnServerProcedure event handler on the database server itself.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.AddRemoteProcedureUser Method

```
void __fastcall AddRemoteProcedureUser(const
      System::UnicodeString ProcedureName, const System::UnicodeString
      AuthorizedUser, TProcedureRights RightsToAssign)
```

Call the AddRemoteProcedureUser method to add rights for an existing user to an existing server-side procedure on a database server. Use the ProcedureName parameter to specify the existing server-side procedure name, the AuthorizedUser parameter to specify the existing user, and the RightsToAssign parameter to specify the rights to give to the user for the procedure. You may use a wildcard (*) for the AuthorizedUser parameter. For example, you could specify just "*" for all users or "Accounting*" for all users whose user name begins with "Accounting".

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.AddRemoteUser Method

```
void __fastcall AddRemoteUser(const System::UnicodeString
      UserName, const System::UnicodeString UserPassword, const
      System::UnicodeString UserDescription, bool IsAdministrator =
      false, System::Word MaxConnections = (System::Word)(0x64))
```

Call the AddRemoteUser method to add a new user to a database server. Use the UserName parameter to specify the new user name, the UserPassword parameter to specify the user's password, the UserDescription parameter to specify a description of the user, the IsAdministrator parameter to indicate whether the new user is an administrator, and the MaxConnections property is used to specify how many concurrent connections this user is allowed to have at any given time.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.CallRemoteProcedure Method

```
void __fastcall CallRemoteProcedure(const System::UnicodeString
      ProcedureName)
```

Call the CallRemoteProcedure method to call a server-side procedure on a database server. Use the ProcedureName parameter to specify which procedure to call. You should populate the RemoteParams property with the desired parameters prior to calling the CallRemoteProcedure method and, once the server-side procedure returns, you can then inspect the RemoteParams property to retrieve the return values, if any, from the server-side procedure.

Note
You must have execute rights to the specified server-side procedure or an error will result.

TDBISAMSession.Close Method

```
void __fastcall Close(void)
```

Call the Close method to close the session and disconnect from a database server if the SessionType property is set to stRemote. The Close method disconnects all active TDBISAMDatabase components that are linked to the session via their SessionName property, which in turn closes all TDBISAMQuery and TDBISAMTable components linked to these databases.

Note
Setting the Active property to False also closes a session.

TDBISAMSession.CloseDatabase Method

```
void __fastcall CloseDatabase(TDBISAMDatabase* Database)
```

Call the CloseDatabase method to close a TDBISAMDatabase component linked to the current session. The Database parameter specifies TDBISAMDatabase component that you wish to close.

The CloseDatabase method decrements the specified TDBISAMDatabase component's reference count and then, if the reference count is zero and the TDBISAMDatabase component's KeepConnection property is False, closes the TDBISAMDatabase component.

TDBISAMSession.DeleteRemoteDatabase Method

```
void __fastcall DeleteRemoteDatabase(const System::UnicodeString
      DatabaseName)
```

Call the DeleteRemoteDatabase method to remove an existing database from a database server. Use the DatabaseName parameter to specify the existing database name.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.DeleteRemoteDatabaseUser Method

```
void __fastcall DeleteRemoteDatabaseUser(const
      System::UnicodeString DatabaseName, const System::UnicodeString
      AuthorizedUser)
```

Call the DeleteRemoteDatabaseUser method to remove rights for an existing user to an existing database on a database server. Use the DatabaseName parameter to specify the existing database name and the AuthorizedUser parameter to specify the existing user.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.DeleteRemoteEvent Method

```
void __fastcall DeleteRemoteEvent(const System::UnicodeString
      EventName)
```

Call the DeleteRemoteEvent method to remove an existing scheduled event from a database server. Use the EventName parameter to specify the existing scheduled event name.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.DeleteRemoteProcedure Method

```
void __fastcall DeleteRemoteProcedure(const
      System::UnicodeString ProcedureName)
```

Call the DeleteRemoteProcedure method to remove an existing server-side procedure from a database server. Use the ProcedureName parameter to specify the existing server-side procedure name.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.DeleteRemoteProcedureUser Method

```
void __fastcall DeleteRemoteProcedureUser(const
      System::UnicodeString ProcedureName, const System::UnicodeString
      AuthorizedUser)
```

Call the DeleteRemoteProcedureUser method to remove rights for an existing user to an existing server-side procedure on a database server. Use the ProcedureName parameter to specify the existing server-side procedure name and the AuthorizedUser parameter to specify the existing user.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.DeleteRemoteUser Method

```
void __fastcall DeleteRemoteUser(const System::UnicodeString
      UserName)
```

Call the DeleteRemoteUser method to remove an existing user from a database server. Use the UserName parameter to specify the existing user name.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.DisconnectRemoteSession Method

```
bool __fastcall DisconnectRemoteSession(int SessionID)
```

Call the DisconnectRemoteSession method to disconnect a specific session on a database server. Disconnecting a session only terminates its connection, it does not remove the session completely from the database server nor does it release any resources for the session other than the thread used for the connection and the connection itself at the operating system level. Use the SessionID parameter to specify the session ID to disconnect. You can get the session ID for a particular session by using the GetRemoteSessionCount and the GetRemoteSessionInfo methods.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.DropConnections Method

```
void __fastcall DropConnections(void)
```

Call the DropConnections method to free all temporary TDBISAMDatabase components for the session that are inactive. If the KeepConnections property of the session is True (the default), then temporary TDBISAMDatabase components created as needed for the session by DBISAM at runtime are not automatically freed when their database connections are closed. DropConnections enables an application to free these TDBISAMDatabase components when they are no longer needed.

TDBISAMSession.FindDatabase Method

```
TDBISAMDatabase* __fastcall FindDatabase(const
      System::UnicodeString DatabaseName)
```

Call the FindDatabase method to searches a session's list of TDBISAMDatabase components for a specified database. The DatabaseName parameter specifies the name of the TDBISAMDatabase component to search for. The FindDatabase method compares the DatabaseName parameter to the DatabaseName property for each TDBISAMDatabase component linked to the session via its SessionName property. If a match is found, the FindDatabase method returns a reference to the TDBISAMDatabase component. Otherwise the FindDatabase method returns nil.

TDBISAMSession.GetDatabaseNames Method

```
void __fastcall GetDatabaseNames(System::Classes::TStrings*
      List)
```

Call the GetDatabaseNames method to populate a string list with the names of all TDBISAMDatabase components linked to the session via their SessionName property. List is a string list object, created and maintained by the application, into which to store the database names.

Note
This method is not the same as the GetRemoteDatabaseNames method, which returns a list of databases defined on a database server.

TDBISAMSession.GetPassword Method

```
bool __fastcall GetPassword(void)
```

Call the GetPassword method to trigger the OnPassword event handler for the session, if one is assigned to the OnPassword event, or display the default password dialog box if an OnPassword event handler is not assigned.

An application can use the return value of the GetPassword method to control program logic. The GetPassword method returns True if a user chooses OK from the default password dialog or the OnPassword event handler sets its Continue parameter to True. The GetPassword method returns False if the user chooses Cancel from the default password dialog or the OnPassword event handler sets its Continue parameter to False.

TDBISAMSession.GetRemoteAdminAddress Method

```
System::UnicodeString __fastcall GetRemoteAdminAddress(void)
```

Call the GetRemoteAdminAddress method to retrieve the IP address that the database server is listening on for administrative connections.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteAdminPort Method

```
int __fastcall GetRemoteAdminPort(void)
```

Call the GetRemoteAdminPort method to retrieve the port that the database server is listening on for administrative connections.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteAdminThreadCacheSize Method

```
int __fastcall GetRemoteAdminThreadCacheSize(void)
```

Call the GetRemoteAdminThreadCacheSize method to retrieve the number of threads that the database server will cache in order to improve connect/disconnect performance for administrative connections.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteConfig Method

```
void __fastcall GetRemoteConfig(bool &DenyLogins, System::Word
      &MaxConnections, System::Word &ConnectTimeout, System::Word
      &DeadSessionInterval, System::Word &DeadSessionExpires,
      System::Word &MaxDeadSessions, System::UnicodeString
      &TempDirectory, System::Classes::TStrings* AuthorizedAddresses,
      System::Classes::TStrings* BlockedAddresses)
```

Call the GetRemoteConfig method to retrieve the current configuration settings for a database server. Please see the ModifyRemoteConfig method for more information on the parameters returned from this method.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server. Also, if the maximum number of connections returned via this method is lower than what was attempted to be configured via the ModifyRemoteConfig method, the ServerLicensedConnections property has caused it to be lowered.

TDBISAMSession.GetRemoteConnectedSessionCount Method

```
int __fastcall GetRemoteConnectedSessionCount(void)
```

Call the GetRemoteConnectedSessionCount method to retrieve the total number of connected sessions on a database server. Sessions that are present on the server, but not connected, are not reported in this figure. To get a total count of the number of sessions on the database server use the GetRemoteSessionCount method instead.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteDatabase Method

```
void __fastcall GetRemoteDatabase(const System::UnicodeString
      DatabaseName, System::UnicodeString &DatabaseDescription,
      System::UnicodeString &ServerPath)
```

Call the GetRemoteDatabase method to retrieve information about an existing database from a database server. Use the DatabaseName parameter to specify the existing database name.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteDatabaseNames Method

```
void __fastcall
      GetRemoteDatabaseNames(System::Classes::TStrings* List)
```

Call the GetRemoteDatabaseNames method to retrieve a list of databases defined on a database server.

Note
Only databases for which the current user has at least drRead rights to will show up in the list populated by the GetRemoteDatabaseNames method. This method is valid for both administrative and regular data connections.

TDBISAMSession.GetRemoteDatabaseUser Method

```
void __fastcall GetRemoteDatabaseUser(const
      System::UnicodeString DatabaseName, const System::UnicodeString
      AuthorizedUser, TDatabaseRights &UserRights)
```

Call the GetRemoteDatabaseUser method to retrieve the rights for an existing user to an existing database on a database server. Use the DatabaseName parameter to specify the existing database name and the AuthorizedUser parameter to specify the existing user.

Note
This method is valid for both administrative and regular data connections. However, for regular data connections this method is only valid if the AuthorizedUser parameter matches that of the CurrentRemoteUser property.

TDBISAMSession.GetRemoteDatabaseUserNames Method

```
void __fastcall GetRemoteDatabaseUserNames(const
      System::UnicodeString DatabaseName, System::Classes::TStrings*
      List)
```

Call the GetRemoteDatabaseUserNames method to retrieve a list of existing users defined with rights for an existing database on a database server. Use the DatabaseName parameter to specify an existing database from which to retrieve a list of users.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteDateTime Method

```
System::TDateTime __fastcall GetRemoteDateTime(void)
```

Call the GetRemoteDateTime method to retrieve the local date and time from a database server.

Note
This method is valid for both administrative and regular data connections.

TDBISAMSession.GetRemoteEngineVersion Method

```
System::UnicodeString __fastcall GetRemoteEngineVersion(void)
```

Call the GetRemoteEngineVersion method to retrieve the DBISAM version from a database server.

Note
This method is valid for both administrative and regular data connections.

TDBISAMSession.GetRemoteEvent Method

```
void __fastcall GetRemoteEvent(const System::UnicodeString
      EventName, System::UnicodeString &EventDescription,
      TEventRunType &EventRunType, System::TDateTime &EventStartDate,
      System::TDateTime &EventEndDate, System::TDateTime
      &EventStartTime, System::TDateTime &EventEndTime, System::Word
      &EventInterval, TEventDays &EventDays, TEventDayOfMonth
      &EventDayOfMonth, TEventDayOfWeek &EventDayOfWeek, TEventMonths
      &EventMonths, System::TDateTime &EventLastRun)
```

Call the GetRemoteEvent method to retrieve information about an existing scheduled event from a database server. Use the EventName parameter to specify the existing event name. Please see the AddRemoteEvent method for more information on the parameters returned from this method. The EventLastRun parameter specifies the last date and time that the event was successfully run.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteEventNames Method

```
void __fastcall GetRemoteEventNames(System::Classes::TStrings*
      List)
```

Call the GetRemoteEventNames method to retrieve a list of scheduled events defined on a database server.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteLogCount Method

```
int __fastcall GetRemoteLogCount(void)
```

Call the GetRemoteLogCount method to retrieve the total count of log records available in the current log on a database server.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteLogRecord Method

```
TLogRecord __fastcall GetRemoteLogRecord(int Number)
```

Call the GetRemoteLogRecord method to retrieve the Nth log record from the current log on a database server.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteMainAddress Method

```
System::UnicodeString __fastcall GetRemoteMainAddress(void)
```

Call the GetRemoteMainAddress method to retrieve the IP address that the database server is listening on for regular data connections.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteMainPort Method

```
int __fastcall GetRemoteMainPort(void)
```

Call the GetRemoteMainPort method to retrieve the port that the database server is listening on for regular data connections.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteMainThreadCacheSize Method

```
int __fastcall GetRemoteMainThreadCacheSize(void)
```

Call the GetRemoteMainThreadCacheSize method to retrieve the number of threads that the database server will cache in order to improve connect/disconnect performance for regular data connections.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteMemoryUsage Method

```
double __fastcall GetRemoteMemoryUsage(void)
```

Call the GetRemoteMemoryUsage method to retrieve the total amount of memory (in megabytes) currently allocated by a database server.

Note
This method has been deprecated and always returns 0 as of version 4.17 of DBISAM and the introduction of the new memory manager used in the DBISAM database server. Please see the Replacement Memory Manager topic for more information.

TDBISAMSession.GetRemoteProcedure Method

```
void __fastcall GetRemoteProcedure(const System::UnicodeString
      ProcedureName, System::UnicodeString &ProcedureDescription)
```

Call the GetRemoteProcedure method to retrieve information about an existing server-side procedure from a database server. Use the ProcedureName parameter to specify the existing server-side procedure name.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteProcedureNames Method

```
void __fastcall
      GetRemoteProcedureNames(System::Classes::TStrings* List)
```

Call the GetRemoteProcedureNames method to retrieve a list of server-side procedures defined on a database server.

Note
Only server-side procedures for which the current user has at least prExecute rights to will show up in the list populated by the GetRemoteProcedureNames method. This method is valid for both administrative and regular data connections.

TDBISAMSession.GetRemoteProcedureUser Method

```
void __fastcall GetRemoteProcedureUser(const
      System::UnicodeString ProcedureName, const System::UnicodeString
      AuthorizedUser, TProcedureRights &UserRights)
```

Call the GetRemoteProcedureUser method to retrieve the rights for an existing user to an existing server-side procedure on a database server. Use the ProcedureName parameter to specify the existing server-side procedure name and the AuthorizedUser parameter to specify the existing user.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteProcedureUserNames Method

```
void __fastcall GetRemoteProcedureUserNames(const
      System::UnicodeString ProcedureName, System::Classes::TStrings*
      List)
```

Call the GetRemoteProcedureUserNames method to retrieve a list of existing users defined with rights for an existing server-side procedure on a database server. Use the ProcedureName parameter to specify an existing server-side procedure from which to retrieve a list of users.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteServerDescription Method

```
System::UnicodeString __fastcall
      GetRemoteServerDescription(void)
```

Use the GetRemoteServerDescription method to retrieve the description of a database server.

Note
This method is valid for both administrative and regular data connections.

TDBISAMSession.GetRemoteServerName Method

```
System::UnicodeString __fastcall GetRemoteServerName(void)
```

Use the GetRemoteServerName method to retrieve the name of a database server.

Note
This method is valid for both administrative and regular data connections.

TDBISAMSession.GetRemoteSessionCount Method

```
int __fastcall GetRemoteSessionCount(void)
```

Call the GetRemoteSessionCount method to retrieve the total number of sessions on a database server. To get a total count of just the number of connected sessions on a database server use the GetRemoteConnectedSessionCount method instead.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteSessionInfo Method

```
bool __fastcall GetRemoteSessionInfo(int SessionNum, int
      &SessionID, System::TDateTime &CreatedOn, System::TDateTime
      &LastConnectedOn, System::UnicodeString &UserName,
      System::UnicodeString &UserAddress, bool &Encrypted,
      System::UnicodeString &LastUserAddress)
```

Call the GetRemoteSessionInfo method to retrieve session information for a specific session on a database server. The SessionNum parameter indicates the session number for which to retrieve the session information. This number represents the logical position of a given session in the list of sessions on a database server, from 1 to the return value of the GetRemoteSessionCount method. The SessionID parameter returns unique ID assigned to the session by the database server. The CreatedOn parameter returns the date and time when the session was created on the database server. The LastConnectedOn parameter returns the date and time when the session was last connected to the database server. The UserName parameter returns the name of the user that created the session on the database server. The UserAddress parameter returns the IP address of the user that created the session on the database server. If the session is not currently connected, then this parameter will be blank. The Encrypted parameter returns whether the session is encrypted or not. The LastUserAddress parameter returns the last known IP address of the session, regardless of whether the session is connected or not. This parameter is useful for determining the location of a workstation that still has an active session but has disconnected.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteUpTime Method

```
__int64 __fastcall GetRemoteUpTime(void)
```

Call the GetRemoteUpTime method to retrieve the number of seconds that the database server has been active and accepting new connections.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteUser Method

```
void __fastcall GetRemoteUser(const System::UnicodeString
      UserName, System::UnicodeString &UserPassword,
      System::UnicodeString &UserDescription, bool &IsAdministrator,
      System::Word &MaxConnections)
```

Call the GetRemoteUser method to retrieve information about an existing user from a database server. Use the UserName parameter to specify the existing user name.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteUserNames Method

```
void __fastcall GetRemoteUserNames(System::Classes::TStrings*
      List)
```

Call the GetRemoteUserNames method to retrieve a list of users defined on a database server.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.GetRemoteUTCDateTime Method

```
System::TDateTime __fastcall GetRemoteUTCDateTime(void)
```

Call the GetRemoteUTCDateTime method to retrieve the universal coordinate date and time from a database server. This is especially useful if you are accessing a database server in a different time zone and wish to get the date and time in a standard format that doesn't need to take into account the local server time offset.

Note
This method is valid for both administrative and regular data connections.

TDBISAMSession.GetTableNames Method

```
void __fastcall GetTableNames(const System::UnicodeString
      DatabaseName, System::Classes::TStrings* List)
```

Call the GetTableNames method to populate a string list with the names of all tables found in the TDBISAMDatabase component specified by the DatabaseName parameter. List is a string list object, created and maintained by the application, into which to store the database names.

Note
The DatabaseName parameter can refer to either the DatabaseName property of a TDBISAMDatabase component or the directory or remote database name of an actual database. If the DatabaseName parameter matches the DatabaseName property of an existing TDBISAMDatabase component, then the table names returned will be from that TDBISAMDatabase component. Otherwise, the DatabaseName parameter will be treated as a directory for a local session and a database name for a remote session and the table names will be retrieved from the appropriate database.

TDBISAMSession.ModifyRemoteConfig Method

```
void __fastcall ModifyRemoteConfig(bool DenyLogins, System::Word
      MaxConnections, System::Word ConnectTimeout, System::Word
      DeadSessionInterval, System::Word DeadSessionExpires,
      System::Word MaxDeadSessions, const System::UnicodeString
      TempDirectory, System::Classes::TStrings* AuthorizedAddresses,
      System::Classes::TStrings* BlockedAddresses)
```

Call the ModifyRemoteConfig method to modify the current configuration settings for a database server. The DenyLogins parameter indicates whether any new logins are denied on the database server. The MaxConnections parameter indicates the maximum allowable number of connected sessions (not total sessions) on the database server. The ConnectTimeout parameter indicates how long a session is allowed to remain idle before the session is disconnected automatically by the database server. The DeadSessionInterval parameter indicates how often the database server should check for dead sessions (sessions that have been disconnected for DeadSessionExpires seconds). The DeadSessionExpires parameter indicates when a disconnected session is considered "dead" based upon the number of seconds since it was last connected. Specifying 0 for this parameter will cause the database server to never consider disconnected sessions as dead and instead will keep them around based upon the MaxDeadSessions parameter alone. The MaxDeadSessions parameter indicates how many dead sessions are allowed on the database server before the database server will start removing dead sessions in oldest-first order. The TempDirectory parameter indicates where temporary tables are stored relative to the database server. This setting is global for all users. The AuthorizedAddresses and BlockedAddresses parameters are lists of IP addresses that specify which IP addresses are allowed or blocked from accessing the database server. Both of these accept the use of a leading * wildcard when specifying IP addresses.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.ModifyRemoteDatabase Method

```
void __fastcall ModifyRemoteDatabase(const System::UnicodeString
      DatabaseName, const System::UnicodeString DatabaseDescription,
      const System::UnicodeString ServerPath)
```

Call the ModifyRemoteDatabase method to modify information about an existing database on a database server. Use the DatabaseName parameter to specify the existing database.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.ModifyRemoteDatabaseUser Method

```
void __fastcall ModifyRemoteDatabaseUser(const
      System::UnicodeString DatabaseName, const System::UnicodeString
      AuthorizedUser, TDatabaseRights RightsToAssign)
```

Call the ModifyRemoteDatabaseUser method to modify the rights for an existing user to an existing database on a database server. Use the DatabaseName parameter to specify the existing database name and the AuthorizedUser parameter to specify the existing user. You may use a wildcard (*) for the AuthorizedUser parameter , such as specifying just '*' for all users or 'Accounting*' for all users whose user name begins with 'Accounting'.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.ModifyRemoteEvent Method

```
void __fastcall ModifyRemoteEvent(const System::UnicodeString
      EventName, const System::UnicodeString EventDescription,
      TEventRunType EventRunType, System::TDateTime EventStartDate,
      System::TDateTime EventEndDate, System::TDateTime EventStartTime,
       System::TDateTime EventEndTime, System::Word EventInterval,
      TEventDays EventDays, TEventDayOfMonth EventDayOfMonth,
      TEventDayOfWeek EventDayOfWeek, const TEventMonths &EventMonths)
```

Call the ModifyRemoteEvent method to modify information about an existing scheduled event on a database server. Use the EventName parameter to specify the existing event name. Please see the AddRemoteEvent method for more information on the parameters returned from this method.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.ModifyRemoteProcedure Method

```
void __fastcall ModifyRemoteProcedure(const
      System::UnicodeString ProcedureName, const System::UnicodeString
      ProcedureDescription)
```

Call the ModifyRemoteProcedure method to modify information about an existing server-side procedure on a database server. Use the ProcedureName parameter to specify the existing server-side procedure.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.ModifyRemoteProcedureUser Method

```
void __fastcall ModifyRemoteProcedureUser(const
      System::UnicodeString ProcedureName, const System::UnicodeString
      AuthorizedUser, TProcedureRights RightsToAssign)
```

Call the ModifyRemoteProcedureUser method to modify the rights for an existing user to an existing server-side procedure on a database server. Use the ProcedureName parameter to specify the existing server-side procedure name and the AuthorizedUser parameter to specify the existing user. You may use a wildcard (*) for the AuthorizedUser parameter , such as specifying just '*' for all users or 'Accounting*' for all users whose user name begins with 'Accounting'.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.ModifyRemoteUser Method

```
void __fastcall ModifyRemoteUser(const System::UnicodeString
      UserName, const System::UnicodeString UserPassword, const
      System::UnicodeString UserDescription, bool IsAdministrator =
      false, System::Word MaxConnections = (System::Word)(0x64))
```

Call the ModifyRemoteUser method to modify information about an existing user on a database server. Use the UserName parameter to specify the existing user.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.ModifyRemoteUserPassword Method

```
void __fastcall ModifyRemoteUserPassword(const
      System::UnicodeString UserName, const System::UnicodeString
      UserPassword)
```

Call the ModifyRemoteUserPassword method to modify the password for the current user logged in to the database server. Use the UserName parameter to specify the current user name. This method is only valid for changing the password for the current user.

Note
This method is only valid for remote sessions connected to the regular data port on a database server. If you are logged in to the administration port and are an administrator, use the ModifyRemoteUser method instead.

TDBISAMSession.Open Method

```
void __fastcall Open(void)
```

Call the Open method to start a session. The Open method starts the session and triggers the OnStartup event for the session. If the SessionType property is set to stRemote, then DBISAM will attempt to connect to the database server specified by the RemoteHost or RemoteAddressand RemotePort or RemoteService properties. If the session can successfully connect to the database server, it will then automatically login to the server using the RemoteUser and RemotePasword properties.

Calling the Close method closes any open datasets, and disconnects active database connections. If the SessionType property is set to stRemote, then the connection to the database server is closed and the user is logged out.

TDBISAMSession.OpenDatabase Method

```
TDBISAMDatabase* __fastcall OpenDatabase(const
      System::UnicodeString DatabaseName)
```

Call the OpenDatabase method to open an existing TDBISAMDatabase component, or create a temporary TDBISAMDatabase component and open it. OpenDatabase calls the FindDatabase method to determine if the DatabaseName parameter corresponds to the DatabaseName property of an existing TDBISAMDatabase component. If it does not, OpenDatabase creates a temporary TDBISAMDatabase component, assigning the DatabaseName parameter to the DatabaseName property. It also assigns the DatabaseName parameter to the Directory property if the session is local or the RemoteDatabase property if the session is remote. Finally, OpenDatabase calls the Open method of the TDBISAMDatabase component.

TDBISAMSession.RemoteParamByName Method

```
TDBISAMParam* __fastcall RemoteParamByName(const
      System::UnicodeString Value)
```

Call the RemoteParamByName method to retrieve a specific TDBISAMParam object by name from the RemoteParams property. The Value parameter indicates the name of the parameter that you wish to retrieve. If there are no parameters with that name, the RemoteParamByName method will raise an exception.

TDBISAMSession.RemoveAllPasswords Method

```
void __fastcall RemoveAllPasswords(void)
```

Call the RemoveAllPasswords method to delete all current database table passwords defined for the session. Subsequent attempts to open encrypted database tables that need a password via the TDBISAMQuery or TDBISAMTable components will fail unless an application first calls the AddPassword method to reestablish the necessary password or password for the session.

TDBISAMSession.RemoveAllRemoteMemoryTables Method

```
void __fastcall RemoveAllRemoteMemoryTables(void)
```

Call the RemoveAllRemoteMemoryTables method to delete all in-memory tables residing on the database server for the current client workstation. DBISAM uses the current computer name (Windows) or terminal ID (Linux) along with the current process ID to isolate in-memory tables between various client processes. This method can be called after a process on a client workstation terminates improperly and leaves in-memory tables on the database server that need to be removed.

Note
This method will attempt to delete all in-memory tables for a given client workstation. It is not specific to a client process, so do not call this method if you have multiple processes running on the same client workstation unless you are prepared to possibly have one process delete another process's in-memory tables. However, any failure to delete an in-memory table because it is in use, etc. is ignored by this method and is not raised as an exception.

TDBISAMSession.RemovePassword Method

```
void __fastcall RemovePassword(const System::UnicodeString
      Password)
```

Call the RemovePassword method to delete a single database table password defined for the session. Subsequent attempts to open any encrypted database table that requires this password via the TDBISAMQuery or TDBISAMTable components will fail unless an application first calls the AddPassword method to reestablish a password.

TDBISAMSession.RemoveRemoteSession Method

```
bool __fastcall RemoveRemoteSession(int SessionID)
```

Call the RemoveRemoteSession method to completely remove a specific session on a database server. Removing a session not only terminates its connection, but it also removes the session completely and releases any resources for the session including the thread used for the connection and the connection itself at the operating system level. Use the SessionID parameter to specify the session ID to disconnect. You can get the session ID for a particular session by using the GetRemoteSessionCount and the GetRemoteSessionInfo methods.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.SendProcedureProgress Method

```
void __fastcall SendProcedureProgress(const
      System::UnicodeString Status, System::Word PercentDone, bool
      &Abort)
```

Call the SendProcedureProgress method to send progress information from a server-side procedure on a database server back to the calling session. When the session receives this progress information, it then triggers the OnRemoteProcedureProgress event. Use the Status parameter to specify status information regarding the progress, the PercentDone parameter to indicate the percentage of progress, and the Abort parameter to allow the session to force an abort of the current processing.

Note
The server-side procedure code can choose to respect the Abort parameter or ignore it. It is strictly up to the developer of the server-side procedure.

TDBISAMSession.StartRemoteServer Method

```
void __fastcall StartRemoteServer(void)
```

Call the StartRemoteServer method to cause the database server to start accepting regular data connections from remote sessions. You may call the StartRemoteServer or StopRemoteServer methods without removing existing sessions. When stopping the server, however, all sessions will be automatically disconnected.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.StopRemoteServer Method

```
void __fastcall StopRemoteServer(void)
```

Call the StopRemoteServer method to cause the database server to stop accepting regular data connections from remote sessions. You may call the StartRemoteServer or StopRemoteServer methods without removing existing sessions. When stopping the server, however, all sessions will be automatically disconnected.

Note
This method is only valid for encrypted remote sessions connected as an administrator to the administration port on a database server.

TDBISAMSession.TDBISAMSession Method

```
__fastcall virtual TDBISAMSession(System::Classes::TComponent*
      AOwner)
```

Use the New operator to create an instance of a TDBISAMSession component using the constructor. The default TDBISAMSession component, represented by the global Session function, is created automatically when the application starts. At design time, create additional sessions by dropping them on a data module.

The constructor also:

- Sets the KeepConnections property to True.
- Adds the session to the TDBISAMEngine component's Sessions property.
- Sets the Handle property to nil.

TDBISAMSession.OnPassword Event

```
__property TPasswordEvent OnPassword
```

The OnPassword event is fired when an application attempts to open an encrypted database table for the first time using a TDBISAMQuery or TDBISAMTable component and DBISAM cannot locate the password in the list of defined passwords for the session. To gain access to the database table, the OnPassword event handler must pass a valid password to DBISAM. The event handler should call the AddPassword method to define the password for the session. If the Continue parameter is set to True by the OnPassword event handler, the table open is attempted again. If the Continue parameter is set to False, DBISAM does not try to open the table again and an exception is raised.

Note
If an OnPassword event handler is not assigned to the OnPassword event, but DBISAM needs a password in order to open an encrypted database table, a default password dialog will be displayed to the user that prompts for a password. However, any version of DBISAM for Delphi 6 or higher (including C++Builder 6 and higher as well as Kylix 2 and higher) requires that you include the DBPWDlg unit to your uses clause in order to enable the display of a default password dialog. This is done to allow for DBISAM to be included in applications without linking in the forms support, which can add a lot of unnecessary overhead and also cause unwanted references to user interface libraries. This is not required for Delphi 5 or C++Builder 5, but these versions always include forms support.

TDBISAMSession.OnRemoteLogin Event

```
__property TLoginEvent OnRemoteLogin
```

The OnRemoteLogin event is fired when the session connects to a database server and the RemoteUser and RemotePassword properties have not been assigned or have been assigned but are not valid for the database server. You can specify the user name and password via the UserName and Password parameters. The Continue parameter indicates whether the connection process should continue or whether the session should stop trying to connect to the database server.

Note
Any version of DBISAM for Delphi 6 or higher (including C++Builder 6 and higher as well as Kylix 2 and higher) requires that you include the DBLogDlg unit to your uses clause in order to enable the display of a default remote login dialog. This is done to allow for DBISAM to be included in applications without linking in the forms support, which can add a lot of unnecessary overhead and also cause unwanted references to user interface libraries. This is not required for Delphi 5 or C++Builder 5, but these versions always include forms support.

TDBISAMSession.OnRemoteProcedureProgress Event

```
__property TProcedureProgressEvent OnRemoteProcedureProgress
```

The OnRemoteProcedureProgress event is fired whenever a server-side procedure that has been called by the current remote session sends a progress notification back to the remote session using the SendProcedureProgress method. The Status parameter is a status message defined by the server-side procedure, the PercentDone parameter indicates the percentage of the server-side procedure currently executed, and the Abort variable parameter allows the remote session to abort the server-side procedure.

Note
Whether the server-side procedure respects the setting of the Abort parameter is completely up to the server-side procedure and how it is implemented. It can choose to ignore this parameter completely, if desired.

TDBISAMSession.OnRemoteReceiveProgress Event

```
__property TSendReceiveProgressEvent OnRemoteReceiveProgress
```

The OnRemoteReceiveProgress event is fired whenever a remote session receives a response from the database server. The NumBytes parameter indicates the amount of data in bytes that has been received so far, and always starts at 0 bytes to indicate the beginning of a response. The PercentDone parameter indicates the percentage of the response that has been received so far, and is also 0 at the beginning of a response.

TDBISAMSession.OnRemoteReconnect Event

```
__property TReconnectEvent OnRemoteReconnect
```

The OnRemoteReconnect event is fired when a remote session tries to send a request to the database server and cannot because the connection to the database server has been broken. This is usually due to network issues or the remote session being disconnected by the database server because the connection timeout setting configured for the database server has been exceeded. In such a case the remote session would normally attempt an automatic reconnection. However, attaching an event handler to this event intercepts this reconnection process and allows the application to choose to skip the automatic reconnection by setting the Continue parameter to False (the default value is True). This can be useful in situations where the application knows that the network is down or there is a configuration issue that would prevent the remote session from reconnecting successfully. The application can also set the StopAsking parameter to True to tell DBISAM that it should stop firing this event from now until the connection is finally terminated. This avoids a lot of calls to the event handler as tables and databases are closed and each of them try to send requests to the database server.

TDBISAMSession.OnRemoteSendProgress Event

```
__property TSendReceiveProgressEvent OnRemoteSendProgress
```

The OnRemoteSendProgress event is fired whenever a remote session sends a request to the database server. The NumBytes parameter indicates the amount of data in bytes that has been sent so far, and always starts at 0 bytes to indicate the beginning of a request. The PercentDone parameter indicates the percentage of the request that has been sent so far, and is also 0 at the beginning of a request.

TDBISAMSession.OnRemoteTimeout Event

```
__property TTimeoutEvent OnRemoteTimeout
```

The OnRemoteTimeout event is fired when a remote session is waiting on a response from a database server and has not received a response within the number of seconds indicated by the RemoteTimeout property. The StayConnected parameter indicates whether the remote session should stay connected and keep waiting on a response or whether it should disconnect from the server.

TDBISAMSession.OnRemoteTrace Event

```
__property TTraceEvent OnRemoteTrace
```

The OnRemoteTrace event is fired when remote message tracing is enabled for a remote session via the RemoteTrace property and a request is being sent to the database server or a response is being received from the database server. You can use the TraceRecord parameter to log information about the request or response.

TDBISAMSession.OnShutdown Event

```
__property System::Classes::TNotifyEvent OnShutdown
```

The OnShutdown event is fired when an application deactivates a session. Assign an event handler to the OnShutdown event to take specific actions when an application deactivates a session. A session is deactivated by setting its Active property to False or calling its Close method.

Note
You should not call the TDBISAMSession Open method or toggle the Active property from within this event handler. Doing so can cause infinite recursion.

TDBISAMSession.OnStartup Event

```
__property System::Classes::TNotifyEvent OnStartup
```

The OnStartup event is fired when an application activates a session. Assign an event handler to the OnStartup event to take specific actions when an application activates a session. A session is activated by setting its Active property to True, calling its Open method, or by opening or activating a TDBISAMDatabase, TDBISAMQuery, or TDBISAMTable component linked to the session via their SessionName properties.

Note
You should not call the TDBISAMSession Open method or toggle the Active property from within this event handler. Doing so can cause infinite recursion.

5.22 TDBISAMSQLUpdateObject Component

Header File: dbisamtb

Inherits From TDBISAMDataSetUpdateObject

The TDBISAMSQLUpdate component is an abstract component that is implemented by the TDBISAMUpdateSQL component. Normally, only developers interested in creating their own custom update components would use the TDBISAMSQLUpdateObject.

Properties	Methods	Events
	TDBISAMSQLUpdateObject	

TDBISAMSQLUpdateObject.TDBISAMSQLUpdateObject Method

```
inline __fastcall virtual
      TDBISAMSQLUpdateObject(System::Classes::TComponent* AOwner) :
      TDBISAMDataSetUpdateObject(AOwner) { }
```

Use the New operator to create an instance of a TDBISAMSQLUpdateObject component using the constructor. However, you should not use this constructor to create an instance of this component directly. Instead you should create an instance of a TDBISAMUpdateSQL component, which descends from this component and provides a published interface for use at design-time.

5.23 TDBISAMStringList Component

Header File: dbisamtb

Inherits From TDBISAMLocaleStringList

Use the TDBISAMStringList component with the TDBISAMEngine BuildWordList method as a string list that sorts its contents using a specific locale.

Properties	Methods	Events
LocaleID	FindPartial	
	FindUsingPartials	
	TDBISAMStringList	

TDBISAMStringList.LocaleID Property

```
__property int LocaleID
```

Use the LocaleID property to specify what locale you wish to use for sorting the contents of the string list. You can use the TDBISAMEngine GetLocaleNames to get a list of locale names and their IDs.

TDBISAMStringList.FindPartial Method

```
virtual bool __fastcall FindPartial(const System::UnicodeString
      S, int &Index, System::Word PartialLength)
```

Use the FindPartial method to find a given string in the string list using only the first N characters indicated by the PartialLength parameter. This method returns True if the string is found in the list and False if it is not. If the string is found, the Index variable parameter will contain the 0-based position of the string in the string list.

TDBISAMStringList.FindUsingPartials Method

```
virtual bool __fastcall FindUsingPartials(const
      System::UnicodeString S, int &Index)
```

Use the FindUsingPartials method to find a given string in the string list where the list may contain strings with wildcard (*) characters. This method returns True if the string is found in the list and False if it is not. If the string is found, the Index variable parameter will contain the 0-based position of the string in the string list.

TDBISAMStringList.TDBISAMStringList Method

```
inline __fastcall virtual TDBISAMStringList(void) :
      Dbisamlb::TDBISAMLocaleStringList() { }
```

Use the New operator to create an instance of a TDBISAMStringList component using the constructor.

5.24 TDBISAMTable Component

Header File: dbisamtb

Inherits From TDBISAMDBDataSet

Use the TDBISAMTable component to access records and fields in an underlying table. A TDBISAMTable component can also work with a subset of records within a table using ranges and filters as well as access both disk-based and in-memory tables.

Properties	Methods	Events
BlobBlockSize	AddIndex	OnAlterProgress
Description	AlterTable	OnCopyProgress
Encrypted	ApplyRange	OnDataLost
EngineVersion	CancelRange	OnExportProgress
Exclusive	CopyTable	OnImportProgress
Exists	CreateTable	OnIndexProgress
FieldDefs	DeleteAllIndexes	OnLoadFromStreamProgress
FullTableName	DeleteIndex	OnOptimizeProgress
IndexDefs	DeleteTable	OnRepairLog
IndexFieldCount	EditKey	OnRepairProgress
IndexFieldNames	EditRangeEnd	OnSaveToStreamProgress
IndexFields	EditRangeStart	OnUpgradeLog
IndexName	EmptyTable	OnUpgradeProgress
IndexPageSize	FindKey	OnVerifyLog
KeyFieldCount	FindNearest	OnVerifyProgress
LastAutoIncValue	GetDetailLinkFields	
LastUpdated	GetIndexNames	
LocaleID	GotoCurrent	
MasterFields	GotoKey	
MasterSource	GotoNearest	
Password	LockSemaphore	
PhysicalRecordCount	LockTable	
Ranged	OptimizeTable	
ReadOnly	RecordIsLocked	
StoreDefs	RenameTable	
TableName	RepairTable	
TableSize	SetKey	

TextIndexFields	SetRange	
TextIndexIncludeChars	SetRangeEnd	
TextIndexSpaceChars	SetRangeStart	
TextIndexStopWords	TableIsLocked	
UserMajorVersion	TDBISAMTable	
UserMinorVersion	UnlockSemaphore	
VersionNum	UnlockTable	
	UpgradeTable	
	VerifyTable	

TDBISAMTable.BlobBlockSize Property

```
__property int BlobBlockSize
```

The BlobBlockSize property indicates the BLOB block size being used for the table.

Note
This property is read-only, you must alter the structure of a table in order to change it. Also, this property does not require that the table be open in order to return a value. This allows the the property to be examined without opening the table.

TDBISAMTable.Description Property

```
__property System::UnicodeString Description
```

The Description property indicates the description of the table. DBISAM allows you to assign a description to any table.

Note
This property is read-only, you must alter the structure of a table in order to change it. Also, this property does not require that the table be open in order to return a value. This allows the the property to be examined without opening the table.

TDBISAMTable.Encrypted Property

```
__property bool Encrypted
```

Indicates whether the current database table is encrypted. If a table is encrypted with a password, you must provide this password before DBISAM will allow you to open the table or access it in any way.

Note
This property is read-only, you must alter the structure of a table in order to change it. Also, this property does not require that the table be open in order to return a value. This allows the the property to be examined without opening the table.

TDBISAMTable.EngineVersion Property

```
__property System::UnicodeString EngineVersion
```

Indicates the current version of DBISAM being used. This property is read-only, but published so that it is visible in the Object Inspector in Delphi, Kylix, and C++Builder.

TDBISAMTable.Exclusive Property

```
__property bool Exclusive
```

Use the Exclusive property to True to specify that a table should be opened exclusively when calling the Open method or when setting the Active property to True. When the Exclusive property is set to True and the application successfully opens the table, no other application can access the table. If the table for which the application has requested exclusive access is already in use by another application, an exception is raised.

A table must be closed (Active property should be False) before changing the setting of the Exclusive property. Do not set Exclusive to True at design time if you also intend to set the Active property to True at design time. In this case an exception is raised because the table is already in use by the IDE.

TDBISAMTable.Exists Property

```
__property bool Exists
```

The Exists property indicates whether the underlying table exists. This property uses the current DatabaseName and TableName properties to determine the location of the table.

TDBISAMTable.FieldDefs Property

```
__property TDBISAMFieldDefs* FieldDefs
```

The FieldDefs property lists the field definitions for a dataset. While an application can examine FieldDefs to explore the field definitions for a table, it should not change these definitions unless creating a new table with the CreateTable method. To access fields and field values in a table, use the Fields property and the FieldByName method. If the FieldDefs property is updated or manually edited, the StoreDefs property is automatically set to True.

Note
The field definitions in the FieldDefs may not always reflect the current field definitions available for a table unless the table has been opened. Before using the field definitions from an existing table for in a call to the AlterTable method, call the Update method to read the field definitions from the actual table.

TDBISAMTable.FullTableName Property

```
__property System::UnicodeString FullTableName
```

The FullTableName property indicates the full table name, including the path, for the table. This read-only property can be examined at runtime to find out the exact physical location of a table on disk, if the session is local, or the logical location of a table on a database server, if the session is remote.

TDBISAMTable.IndexDefs Property

```
__property TDBISAMIndexDefs* IndexDefs
```

The IndexDefs property lists the index definitions for a dataset. While an application can examine IndexDefs to explore the index definitions for a table, it should not change these definitions unless creating a new table with the CreateTable method. To set the active index for a table, use the IndexName or IndexFieldNames property. If the IndexDefs property is updated or manually edited, the StoreDefs property is automatically set to True.

Note
The index definitions in the IndexDefs may not always reflect the current index definitions available for a table unless the table has been opened and the IndexName or IndexFieldNames property has been assigned a value. Before using the index definitions from an existing table for in a call to the AlterTable method, call the Update method to read the index definitions from the actual table.

TDBISAMTable.IndexFieldCount Property

```
__property int IndexFieldCount
```

The IndexFieldCount property indicates the number of fields that make up the active index. The IndexName or IndexFieldNames property can be used to set and inspect the active index for the table.

TDBISAMTable.IndexFieldNames Property

```
__property System::UnicodeString IndexFieldNames
```

Use the IndexFieldNames property as an alternative method to the IndexName property of specifying the active index for a table. Each column name should be separated with a semicolon. Any column names specified in the IndexFieldNames property must already be indexed, and must exist in the index in the order specified, from left to right.

Note
The IndexFieldNames and IndexName properties are mutually exclusive. Setting one clears the other.

TDBISAMTable.IndexFields Property

```
__property Data::Db::TField* IndexFields[int Index]
```

Use the IndexFields property to access a TField object for a given field in an index. The IndexFields property provides a zero-based array of TField objects. The first field in the index is referenced as IndexFields[0], the second is referenced as IndexFields[1], and so on.

Note
Do not set the IndexFields propety directly. Instead use the IndexName or IndexFieldNames property to set the active index for a table.

TDBISAMTable.IndexName Property

```
__property System::UnicodeString IndexName
```

Use the IndexName property to specify the active index for a table. If the IndexName property is empty (the default), the active index is set to the primary index. If the IndexName property is set to a valid index name, then that index is used to determine the sort order of records, otherwise an exception will be raised.

Note
The IndexName and IndexFieldNames properties are mutually exclusive. Setting one clears the other.

TDBISAMTable.IndexPageSize Property

```
__property int IndexPageSize
```

The IndexPageSize property indicates the index page size being used for the table.

Note
This property is read-only, you must alter the structure of a table in order to change it. Also, this property does not require that the table be open in order to return a value. This allows the the property to be examined without opening the table.

TDBISAMTable.KeyFieldCount Property

```
__property int KeyFieldCount
```

Use the KeyFieldCount property to limit a search on the active multi-field index to a consecutive sub-set (left to right) of those fields. For example, if the active index for a table consists of three fields, a partial-key search can be conducted using only the first field in the index by setting KeyFieldCount to 1. If the KeyFieldCount property is 0, the table searches on all fields in the index. The active index for a table is specified via the IndexName or IndexFieldNames property.

Note
Searches are only conducted based on consecutive key fields beginning with the first field in the key. For example if an index consists of three fields, an application can set the KeyFieldCount property 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.

TDBISAMTable.LastAutoIncValue Property

```
__property int LastAutoIncValue
```

The LastAutoIncValue property indicates the last auto-increment value used for the table. This read-only property can be examined to determine what value will be assigned to the next auto-increment field when a record is appended to the table.

Note
This property is read-only, you must alter the structure of a table in order to change it. Also, this property does not require that the table be open in order to return a value. This allows the the property to be examined without opening the table.

TDBISAMTable.LastUpdated Property

```
__property System::TDateTime LastUpdated
```

The LastUpdated property indicates the last date and time the table was modified. The LastUpdated property is maintained automatically by DBISAM whenever an update occurs on the table.

Note
The table must be open (Active=True) before the LastUpdated property can be accessed.

TDBISAMTable.LocaleID Property

```
__property int LocaleID
```

The LocaleID property indicates the locale being used for the table.

Note
This property is read-only, you must alter the structure of a table in order to change it. Also, this property does not require that the table be open in order to return a value. This allows the the property to be examined without opening the table.

TDBISAMTable.MasterFields Property

```
__property System::UnicodeString MasterFields
```

After setting the MasterFields property, use the MasterFields property to specify the names of one or more fields to use in establishing a master-detail link between this table and the data source specified in the MasterSource property. Separate multiple field names with a semicolon. Each time the current record in the master data source changes, the new values in the master fields are used to select corresponding records in this table for display.

Note
At design time, you can use the Field Link property editor to establish a master-detail link between a data source and the current table.

TDBISAMTable.MasterSource Property

```
__property Data::Db::TDataSource* MasterSource
```

Use the MasterSource property to specify the name of a TDataSource component whose DataSet property identifies a dataset to use as a master table in establishing a master-detail link with this table. After setting the MasterSource property, specify which fields to use in the master data source by setting the MasterFields property.

TDBISAMTable.Password Property

```
__property System::UnicodeString Password
```

The Password property indicates the password used to encrypt the table, if it is encrypted, or a blank string if it is not encrypted. If a table is encrypted with a password, you must provide this password before DBISAM will allow you to open the table or access it in any way.

Note
This property is read-only, you must alter the structure of a table in order to change it. Also, this property does not require that the table be open in order to return a value. This allows the the property to be examined without opening the table.

TDBISAMTable.PhysicalRecordCount Property

```
__property int PhysicalRecordCount
```

The PhysicalRecordCount property indicates the number of records present in the table, irrespective of any filters or ranges that may currently be active.

TDBISAMTable.Ranged Property

```
__property bool Ranged
```

The Ranged property indicates whether a range if active for the current table.

TDBISAMTable.ReadOnly Property

```
__property bool ReadOnly
```

Use the ReadOnly property to prevent any updates in the table. The default value is False, meaning users can insert, update, and delete data in the table. When the ReadOnly property is True, the table's CanModify property is False.

Note
If, due to the security configuration of the operating system or the database server, the table cannot be opened in a read-write fashion, the ReadOnly property will automatically be set to True when the table is opened.

TDBISAMTable.StoreDefs Property

```
__property bool StoreDefs
```

The StoreDefs property indicates whether the FieldDefs property and its contained list of TDBISAMFieldDef objects, as well as the IndexDefs property and its contained list of TDBISAMIndexDef objects, will be stored at design-time. This is especially useful for in-memory tables that do not actually have a physical table structure on disk, and will eliminate having to constantly specify this information prior to calling the CreateTable method.

Note
It is not recommended that you set this property to True for disk-based tables.

TDBISAMTable.TableName Property

```
__property System::UnicodeString TableName
```

Use the TableName property to specify the name of the table this TDBISAMTable component should access. The TableName property is used in conjunction with the DatabaseName property to specify the location and name of the table.

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

TDBISAMTable.TableSize Property

```
__property __int64 TableSize
```

The TableSize property indicates the total size, in bytes, that the table consumes in space on the storage medium where it is stored.

TDBISAMTable.TextIndexFields Property

```
__property System::UnicodeString TextIndexFields
```

The TextIndexFields property indicates the string or memo fields that are currently part of the full text index for the table.

Note
This property is read-only, you must alter the structure of a table in order to change it. Also, this property does not require that the table be open in order to return a value. This allows the the property to be examined without opening the table.

TDBISAMTable.TextIndexIncludeChars Property

```
__property System::UnicodeString TextIndexIncludeChars
```

The TextIncludeChars property indicates the include characters used for building the full text index for the table. Include characters are used to determine how words are formed by specifying which characters are included in the word and which are ignored.

Note
This property is read-only, you must alter the structure of a table in order to change it. Also, this property does not require that the table be open in order to return a value. This allows the the property to be examined without opening the table.

TDBISAMTable.TextIndexSpaceChars Property

```
__property System::UnicodeString TextIndexSpaceChars
```

The TextIndexSpaceChars property indicates the space characters used for building the full text index for the table. Space characters are used to determine how words are separated from one another for the purposes of indexing the words in the full text index.

Note
This property is read-only, you must alter the structure of a table in order to change it. Also, this property does not require that the table be open in order to return a value. This allows the the property to be examined without opening the table.

TDBISAMTable.TextIndexStopWords Property

```
__property System::Classes::TStrings* TextIndexStopWords
```

The TextIndexStopWords property indicates the stop words used for building the full text index for the table. Stop words are words that will be removed from the index due to the fact that they are too common, such as is the case with the word "the".

Note
This property is read-only, you must alter the structure of a table in order to change it. Also, this property does not require that the table be open in order to return a value. This allows the the property to be examined without opening the table. Finally, this property is a TStrings object, which by the nature of the property behavior in Delphi, Kylix, or C++Builder can be modified. However, these modifications will not be saved and the property will revert to the correct list of stop words for the table whenever the property is accessed.

TDBISAMTable.UserMajorVersion Property

```
__property System::Word UserMajorVersion
```

The UserMajorVersion property indicates the user-defined major version number of the table. DBISAM allows you to assign a user-defined major and minor version number to any table that can aid the database developer in tracking application-specific structure information. This is especially useful for determining whether a table has the correct structure for the current release of an application. The minor version number can be retrieved using the UserMinorVersion property.

Note
This property is read-only, you must alter the structure of a table in order to change it. Also, this property does not require that the table be open in order to return a value. This allows the the property to be examined without opening the table.

TDBISAMTable.UserMinorVersion Property

```
__property System::Word UserMinorVersion
```

The UserMinorVersion property indicates the user-defined minor version number of the table. DBISAM allows you to assign a user-defined major and minor version number to any table that can aid the database developer in tracking application-specific structure information. This is especially useful for determining whether a table has the correct structure for the current release of an application. The major version number can be retrieved using the UserMajorVersion property.

Note
This property is read-only, you must alter the structure of a table in order to change it. Also, this property does not require that the table be open in order to return a value. This allows the the property to be examined without opening the table.

TDBISAMTable.VersionNum Property

```
__property System::UnicodeString VersionNum
```

The VersionNum property indicates the internal format version number of the table. DBISAM assigns each table an internal format version number so that it can make sure that an application does not try to open a table using a different format than what DBISAM expects. The format version number changes whenever the internal format of the DBISAM tables change.

The following is a list of all of the version numbers returned by the VersionNum property:

'1.00'
'1.02'
'1.04'
'1.08'
'2.00'
'3.00'
'4.00' (current)

You'll notice that the most current version of DBISAM does not always coincide with the latest table format version number.

Note
This property does not require that the table be open in order to return a value. This allows the the property to be examined without opening the table.

TDBISAMTable.AddIndex Method

```
void __fastcall AddIndex(const System::UnicodeString Name, const
      System::UnicodeString Fields, Data::Db::TIndexOptions Options =
      Data::Db::TIndexOptions() , const System::UnicodeString
      DescFields = System::UnicodeString(), TIndexCompression
      Compression = (TIndexCompression)(0x0), bool NoKeyStatistics =
      false)
```

Call the AddIndex method to create a new index for the table. The table may be open or closed when executing this method. If the table is open, then is must have been opened exclusively (Exclusive=True) or an exception will be raised.

The Name parameter is the name of the new index. The Fields parameter is a semicolon-delimited list of the fields to include in the index. The Options parameter is a potentially restricted set of attributes for the index:

Option	Description
ixPrimary	Represents the primary index for a table.
ixUnique	Represents a unique index which does not permit duplicate index key values. The primary index is always implicitly unique.
ixDescending	Represents an index that sorts some or all fields in the index in descending order. The DescFields parameter controls which fields.
ixCaseInsensitive	Represents an index that sorts without case-sensitivity.

The DescFields parameter indicates if you wish only certain fields in the index to sort in descending order. The Compression parameter indicates what type of index compression should be used for the new index.

TDBISAMTable.AlterTable Method

```
void __fastcall AlterTable(int NewLocaleID = 0x0, System::Word
      NewUserMajorVersion = (System::Word)(0x1), System::Word
      NewUserMinorVersion = (System::Word)(0x0), bool NewEncrypted =
      false, const System::UnicodeString NewPassword =
      System::UnicodeString(), const System::UnicodeString
      NewDescription = System::UnicodeString(), int NewIndexPageSize =
      0x1000, int NewBlobBlockSize = 0x200, int NewLastAutoIncValue =
      0xffffffff, const System::UnicodeString NewTextIndexFields =
      System::UnicodeString(), System::Classes::TStrings*
      NewTextIndexStopWords = (System::Classes::TStrings*)(0x0), const
      System::UnicodeString NewTextIndexSpaceChars =
      System::UnicodeString(), const System::UnicodeString
      NewTextIndexIncludeChars = System::UnicodeString(), bool
      SuppressBackups = false)
```

Call the AlterTable method to alter the structure of a table using the field definitions and index definitions specified in the FieldDefs and IndexDefs properties, respectively.

The NewLocaleID parameter specifies the new locale ID for the table. If this parameter is 0, then the table will use the default locale of "ANSI Standard".

The NewUserMajorVersion and NewUserMinorVersion parameters specify the new user-defined major and minor version numbers for the table.

The NewEncrypted parameter specifies whether the table should be encrypted, and the NewPassword parameter specifies the specifies the password for the encryption.
The NewDescription parameter specifies the new description for the table.

The NewIndexPageSize and NewBlobBlockSize parameters specify the index page size and BLOB block size for the table, respectively. Please see Appendix C - System Capacities for more information on the proper values for these parameters.

The NewLastAutoIncValue parameter specifies the new last autoinc value for the table.

The NewTextIndexFields is a list of field names that should be included in the full text index for the table. These field names should be separated by semicolons and should only be the names of string or memo fields. Leaving this parameter blank will remove any entries in the full text index. There is no explicit limit to the number of string or memo fields that can be text indexed.

The NewTextIndexStopWords parameter specifies a list of stop words to be used for the full text index. Stop words are words that will be removed from the index due to the fact that they are too common, such as is the case with the word "the". This parameter is a TStrings object, and if you leave this parameter nil DBISAM will use the default stop words list for the full text index.

The NewTextIndexSpaceChars parameter specifies a set of characters to be used as word separators for the full text index. Space characters are used to determine how words are separated from one another for the purposes of indexing the words in the full text index. This parameter is a string, and if you leave this parameter blank DBISAM will use the default space characters above for the full text index.

The NewTextIndexIncludeChars parameter specifies a set of characters to be used as valid characters in a word for the full text index. Include characters are used to determine how words are formed by specifying

which characters are included in the word and which are ignored. This parameter is a string, and if you leave this parameter blank DBISAM will use the default include characters above for the full text index.

> **Note**
> You can retrieve the default full text indexing parameters using the TDBISAMEngine GetDefaultTextIndexParams method. Please see the Full Text Indexing topic for more information.

The SuppressBackups parameter specifies whether the creation of backup files should take place during the alteration of the table structure.

TDBISAMTable.ApplyRange Method

```
void __fastcall ApplyRange(void)
```

Call the ApplyRange method to cause a range established with the SetRangeStart and SetRangeEnd or EditRangeStart and EditRangeEnd methods to take effect. When a range is in effect, only those records that fall within the range are available for viewing and editing.

TDBISAMTable.CancelRange Method

```
void __fastcall CancelRange(void)
```

Call the CancelRange method to remove a range currently applied to a table using the SetRange or ApplyRange methods. Cancelling a range reenables access to all records in the table.

TDBISAMTable.CopyTable Method

```
void __fastcall CopyTable(const System::UnicodeString
      NewDatabaseName, const System::UnicodeString NewTableName, bool
      CopyData = true)
```

Call the CopyTable method to copy the contents of the table to another new table. The NewDatabaseName parameter specifies the new database directory, for local sessions, or database name, for remote sessions, in which to copy the table. The NewTable parameter specifies the name of the table to create. The CopyData parameter indicates whether the data in the table should be copied, and defaults to True. The table may be open or closed when executing this method. If the table is open, then this method will respect any active filters or ranges on the table when copying the data to the new table.

Note
This method will overwrite any existing table with the same name in the specified destination database without raising an exception, so please be careful when using this method.

TDBISAMTable.CreateTable Method

```
void __fastcall CreateTable(int NewLocaleID = 0x0, System::Word
      NewUserMajorVersion = (System::Word)(0x1), System::Word
      NewUserMinorVersion = (System::Word)(0x0), bool NewEncrypted =
      false, const System::UnicodeString NewPassword =
      System::UnicodeString(), const System::UnicodeString
      NewDescription = System::UnicodeString(), int NewIndexPageSize =
      0x1000, int NewBlobBlockSize = 0x200, int NewLastAutoIncValue =
      0xffffffff, const System::UnicodeString NewTextIndexFields =
      System::UnicodeString(), System::Classes::TStrings*
      NewTextIndexStopWords = (System::Classes::TStrings*)(0x0), const
      System::UnicodeString NewTextIndexSpaceChars =
      System::UnicodeString(), const System::UnicodeString
      NewTextIndexIncludeChars = System::UnicodeString())
```

Call the CreateTable method to create a table using the field definitions and index definitions specified in the FieldDefs and IndexDefs properties, respectively. CreateTable will not overwrite an existing table and will instead raise an exception if it encounters an existing table with the same name. To avoid this error, check the Exists property before calling the CreateTable method.

The NewLocaleID parameter specifies the new locale ID for the table. If this parameter is 0, then the table will use the default locale of "ANSI Standard".

The NewUserMajorVersion and NewUserMinorVersion parameters specify the new user-defined major and minor version numbers for the table.

The NewEncrypted parameter specifies whether the table should be encrypted, and the NewPassword parameter specifies the specifies the password for the encryption.

The NewDescription parameter specifies the new description for the table.

The NewIndexPageSize and NewBlobBlockSize parameters specify the index page size and BLOB block size for the table, respectively. Please see Appendix C - System Capacities for more information on the proper values for these parameters.

The NewLastAutoIncValue parameter specifies the new last autoinc value for the table.

The NewTextIndexFields is a list of field names that should be included in the full text index for the table. These field names should be separated by semicolons and should only be the names of string or memo fields. Leaving this parameter blank will remove any entries in the full text index. There is no explicit limit to the number of string or memo fields that can be text indexed.

The NewTextIndexStopWords parameter specifies a list of stop words to be used for the full text index. Stop words are words that will be removed from the index due to the fact that they are too common, such as is the case with the word "the". This parameter is a TStrings object, and if you leave this parameter nil DBISAM will use the default stop words list for the full text index.

The NewTextIndexSpaceChars parameter specifies a set of characters to be used as word separators for the full text index. Space characters are used to determine how words are separated from one another for the purposes of indexing the words in the full text index. This parameter is a string, and if you leave this parameter blank DBISAM will use the default space characters above for the full text index.

The NewTextIndexIncludeChars parameter specifies a set of characters to be used as valid characters in a word for the full text index. Include characters are used to determine how words are formed by specifying which characters are included in the word and which are ignored. This parameter is a string, and if you leave this parameter blank DBISAM will use the default include characters above for the full text index.

Note
You can retrieve the default full text indexing parameters using the TDBISAMEngine GetDefaultTextIndexParams method. Please see the Full Text Indexing topic for more information.

TDBISAMTable.DeleteAllIndexes Method

```
void __fastcall DeleteAllIndexes(void)
```

Call the DeleteAllIndexes method to remove all indexes from the table and cause the table to revert to using the natural record order. The table may be open or closed when executing this method. If the table is open, then is must have been opened exclusively (Exclusive=True) or an exception will be raised.

TDBISAMTable.DeleteIndex Method

```
void __fastcall DeleteIndex(const System::UnicodeString Name)
```

Call the DeleteIndex method to remove a secondary index for a table. The Name parameter is the name of the index to delete. Leave this parameter blank to delete the primary index for the table. The table may be open or closed when executing this method. If the table is open, then is must have been opened exclusively (Exclusive=True) or an exception will be raised.

TDBISAMTable.DeleteTable Method

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

Call the DeleteTable method to delete an existing table. A table must be closed before this method can be called.

Note
Deleting a table erases any data the table contains and destroys the table's structure information.

TDBISAMTable.EditKey Method

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

Call the EditKey method to put the table in dsSetKey state while preserving the current contents of the current search key buffer. To set the current search values, you can use the IndexFields property to iterate over the fields used by the active index. The IndexName or IndexFieldNames property specifies the active index. Once the search values are set, you can then use the GotoKey or GotoNearest method to perform the actual search.

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

TDBISAMTable.EditRangeEnd Method

```
void __fastcall EditRangeEnd(void)
```

Call the EditRangeEnd method to change the ending value for an existing range. To specify an end range value, call the FieldByName method after calling the EditRangeEnd method. After assigning a new ending value, call the ApplyRange method to activate the modified range.

TDBISAMTable.EditRangeStart Method

```
void __fastcall EditRangeStart(void)
```

Call the EditRangeStart method to change the starting value for an existing range. To specify a starting range value, call the FieldByName method after calling the EditRangeStart method. After assigning a new starting value, call the ApplyRange method to activate the modified range.

TDBISAMTable.EmptyTable Method

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

Call the EmptyTable method to delete all records from the table specified by the DatabaseName and TableName properties. The table may be open or closed when executing this method. If the table is open, then is must have been opened exclusively (Exclusive=True) or an exception will be raised.

TDBISAMTable.FindKey Method

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

Call the FindKey method to search for a specific record in a table using the active index. The IndexName or IndexFieldNames property specifies the active index. The KeyValues parameter contains a comma-delimited array of field values. Each value in the KeyValues parameter can be a literal, a variable, a null, or nil. If the number of values passed in the KeyValues parameters is less than the number of columns in the active index, the missing values are assumed to be null. If a search is successful, the FindKey method positions the table on the matching record and returns True. Otherwise the current table position is not altered, and FindKey returns False.

TDBISAMTable.FindNearest Method

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

Call the FindNearest method search for a record in the table that is greater than or equal to the values specified in the KeyValues parameter using the active index. The IndexName or IndexFieldNames property specifies the active index. The KeyValues parameter contains a comma-delimited array of field values. If the number of values passed in the KeyValues parameter is less than the number of columns in the active index, the missing values are assumed to be null. FindNearest positions the table 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.

TDBISAMTable.GetDetailLinkFields Method

```
virtual void __fastcall
      GetDetailLinkFields(System::Classes::TList* MasterFields,
      System::Classes::TList* DetailFields)
```

Call the GetDetailLinkFields method to fill two lists of TField objects that define a master-detail link between this table and another master datasource. The MasterFields parameter is filled with fields from the master data source whose values must equal the values of the fields in the DetailFields parameter. The DetailFields parameter is filled with fields from the table.

TDBISAMTable.GetIndexNames Method

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

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

TDBISAMTable.GotoCurrent Method

```
void __fastcall GotoCurrent(TDBISAMTable* Table)
```

Call the GotoCurrent method to synchronize the current position for the table based on the current position in another table TDBISAMTable component, but which is connected to the same underlying table. The Table parameters is the name of the TDBISAMTable component whose position should be used for synchronizing.

Note
This procedure works only for TDBISAMTable components that have the same DatabaseName and TableName properties. Otherwise an exception is raised.

TDBISAMTable.GotoKey Method

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

Use the GotoKey method to move to a record specified by search values assigned with previous calls to the SetKey or EditKey methods. The search is peformed using the active index. The IndexName or IndexFieldNames property specifies the active index. If the GotoKey method finds a matching record, it positions the table on the record and returns True. Otherwise the current table position remains unchanged, and GotoKey returns False.

TDBISAMTable.GotoNearest Method

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

Call the GotoNearest method to position the table on the record that is either the exact record specified by the current search values, or on the first record whose values exceed those specified. The search is peformed using the active index. The IndexName or IndexFieldNames property specifies the active index. Before calling the GotoNearest method, an application must specify the search values by calling the SetKey or EditKey methods, which put the table into the dsSetKey state. The application then uses the FieldByName method to populate the search values.

TDBISAMTable.LockSemaphore Method

```
bool __fastcall LockSemaphore(int Value)
```

Call the LockSemaphore method to lock the semaphore specified by the Value paramater in the table. You may reference any semaphore value from 1-1024 in the Value parameter. DBISAM allows the database developer to use semaphores on tables to convey concurrency information not available through normal locking methods. For example, it allows for synchronization of different batch processes that may require exclusive access to a given table.

TDBISAMTable.LockTable Method

```
void __fastcall LockTable(void)
```

Call the LockTable method to lock all of the records in a table and prevent other sessions from placing a record lock or table lock on the table.

TDBISAMTable.OptimizeTable Method

```
void __fastcall OptimizeTable(const System::UnicodeString
      OptimizeIndexName = System::UnicodeString(), bool
      SuppressBackups = false)
```

Call the OptimizeTable method to optimize the physical organization of a table according to a specified index, and permanently remove any free space from the table. Use the OptimizeIndexName parameter to specify what index should be used to physically reorder the records in the table. Use the SuppressBackups parameter to suppress the creation of backup files for the physical files that make up a table on disk. The backup files are created in the same physical location as the source table being optimized. The table must be closed when executing this method.

Note
Under normal circumstances you should specify the primary index (blank string "") as the index to use for reorganization of the physical records on disk since this is the order most commonly used for SQL joins and the default display of data. Reorganizing the data in this manner will improve read-ahead buffering in DBISAM, thus improving overall performance of SQL and navigation using the index specified for the optimization.

TDBISAMTable.RecordIsLocked Method

```
bool __fastcall RecordIsLocked(void)
```

Call the RecordIsLocked method to determine if the current record in the table is locked by the current session.

Note
This method only indicates whether the current session has the record locked and does not indicate whether other sessions have the record locked.

TDBISAMTable.RenameTable Method

```
void __fastcall RenameTable(const System::UnicodeString
      NewTableName)
```

Call the RenameTable method to give a new name to a table. A table must be closed before this method can be called.

TDBISAMTable.RepairTable Method

```
bool __fastcall RepairTable(bool ForceIndexRebuild = false)
```

Call the RepairTable method to repair any damage or corruption that may occur to a table due to an improper operating system or application shutdown. This method will return True if the table is okay and False if there is any damage or corruption present in the table. A table must be closed before this method can be called.

The optional ForceIndexRebuild parameter indicates that you wish to have the RepairTable method rebuild the indexes for the current table regardless of whether the RepairTable method finds any corruption in the indexes. In some rare cases of corruption you may need to pass True for this parameter to fix corruption in the indexes for a table that DBISAM cannot detect.

TDBISAMTable.SetKey Method

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

Call the SetKey method to put the table into dsSetKey state and clear the current search values. The FieldByName method can then be used to supply a new set of search values prior to conducting a search using the active index. The IndexName or IndexFieldNames property specifies the active index.

Note
To modify existing search values, call the EditKey method instead.

TDBISAMTable.SetRange Method

```
void __fastcall SetRange(System::TVarRec const *StartValues,
      const int StartValues_Size, System::TVarRec const *EndValues,
      const int EndValues_Size)
```

Call the SetRange method to specify a range and apply it to the table. A range is set using the active index. The IndexName or IndexFieldNames property specifies the active index. The StartValues parameter indicates the field values that designate the first record in the range. The EndValues parameter indicates the field values that designate the last record in the range. If either the StartValues or EndValues parameters has fewer elements than the number of fields in the active index, then the remaining entries are set to NULL.

The SetRange method combines the functionality of the SetRangeStart, SetRangeEnd, and ApplyRange methods in a single method call.

TDBISAMTable.SetRangeEnd Method

```
void __fastcall SetRangeEnd(void)
```

Call the SetRangeEnd method to put the table into dsSetKey state, erase any previous end range values, and set them to NULL. The FieldByName method can be used to set the ending values for a range.

After assigning ending range values to FieldValues, call the ApplyRange method to activate the modified range.

TDBISAMTable.SetRangeStart Method

```
void __fastcall SetRangeStart(void)
```

Call the SetRangeStart method to put the table into dsSetKey state, erase any previous start range values, and set them to NULL. The FieldByName method can be used to set the starting values for a range.

After assigning starting range values to FieldValues, call the ApplyRange method to activate the modified range.

TDBISAMTable.TableIsLocked Method

```
bool __fastcall TableIsLocked(void)
```

Call the TableIsLocked method to determine if the table is locked by the current session.

Note
This method only indicates whether the current session has the table locked and does not indicate whether other sessions have the table locked.

TDBISAMTable.TDBISAMTable Method

```
__fastcall virtual TDBISAMTable(System::Classes::TComponent*
      AOwner)
```

Use the New operator to create an instance of a TDBISAMTable component using the constructor.

TDBISAMTable.UnlockSemaphore Method

```
bool __fastcall UnlockSemaphore(int Value)
```

Call the UnlockSemaphore method to unlock the semaphore specified by the Value parameter for the table. The semaphore should have been locked previously using the LockSemaphore method.

TDBISAMTable.UnlockTable Method

```
void __fastcall UnlockTable(void)
```

Call the UnlockTable method to unlock the table. The table should have been locked previously using the LockTable method.

TDBISAMTable.UpgradeTable Method

```
void __fastcall UpgradeTable(void)
```

Call the UpgradeTable method to upgrade a table from a previous DBISAM table format to the most recent format. Occasionally the format used for tables is changed in order to introduce new features or to improve performance and this method provides and easy way for the developer to transparently upgrade tables to the new table format. A table must be closed before this method can be called.

TDBISAMTable.VerifyTable Method

```
bool __fastcall VerifyTable(void)
```

Call the VerifyTable method to verify a table and see if there is any damage or corruption that may occurred in a table due to an improper operating system or application shutdown. This method will return True if the table is okay and False if there is any damage or corruption present in the table. A table must be closed before this method can be called.

TDBISAMTable.OnAlterProgress Event

```
__property TProgressEvent OnAlterProgress
```

The OnAlterProgress event is fired when the structure of a table is altered using the AlterTable method. Use the PercentDone parameter to display progress information in your application while the table's structure is being altered.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMTable.OnCopyProgress Event

```
__property TProgressEvent OnCopyProgress
```

The OnCopyProgress event is fired when a table is copied to a new table name using the CopyTable method. Use the PercentDone parameter to display progress information in your application while the table is copied.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMTable.OnDataLost Event

```
__property TDataLostEvent OnDataLost
```

The OnDataLost event is fired when using the AlterTable or AddIndexmethod and a change in the structure of the table has caused data to be lost or the addition of a unique index has caused a key violation.

The Cause parameter allows you to determine the cause of the data loss.

The ContextInfo parameter allows you to determine the exact field, index, or table name that is causing or involved in the loss of data.

The Continue parameter allows you to abort the table structure alteration of index addition process and return the table to it's original state with all of the data intact.

The StopAsking parameter allows you to tell DBISAM to stop reporting data loss problems and simply complete the operation.

Note
You may set the Continue parameter to True several times and at a later time set the Continue parameter to False and still have the table retain its original content and structure.

TDBISAMTable.OnExportProgress Event

```
__property TProgressEvent OnExportProgress
```

The OnExportProgress event is fired when a table is exported to a text file using the ExportTable method. Use the PercentDone parameter to display progress information in your application while the table is exported.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMTable.OnImportProgress Event

```
__property TProgressEvent OnImportProgress
```

The OnImportProgress event is fired when a table is imported from a text file using the ImportTable method. Use the PercentDone parameter to display progress information in your application while the table is imported.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMTable.OnIndexProgress Event

```
__property TProgressEvent OnIndexProgress
```

The OnIndexProgress event is fired when a new index is added to a table using the AddIndex method. Use the PercentDone parameter to display progress information in your application while the index is being added.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMTable.OnLoadFromStreamProgress Event

```
__property TProgressEvent OnLoadFromStreamProgress
```

The OnLoadFromStreamProgress event is fired when a stream is loaded into a table using the LoadFromStream method. Use the PercentDone parameter to display progress information in your application while the table is being loaded from the stream.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMTable.OnOptimizeProgress Event

```
__property TProgressEvent OnOptimizeProgress
```

The OnOptimizeProgress event is fired when a table is optimized using the OptimizeTable method. Use the PercentDone parameter to display progress information in your application while the table is being optimized.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMTable.OnRepairLog Event

```
__property TLogEvent OnRepairLog
```

The OnRepairLog event is fired when a table is repaired using the RepairTable method and DBISAM needs to indicate the current status of the repair (such as the start or finish) or an error is found in the integrity of the table. Use the LogMesssage parameter to display repair log information in your application while the table is being repaired or to save the log messages to a file for later viewing.

TDBISAMTable.OnRepairProgress Event

```
__property TSteppedProgressEvent OnRepairProgress
```

The OnRepairProgress event is fired when a table is repaired using the RepairTable method. Use the Step and PercentDone parameters to display progress information in your application while the table is being repaired.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMTable.OnSaveToStreamProgress Event

```
__property TProgressEvent OnSaveToStreamProgress
```

The OnSaveToStreamProgress event is fired when a table is saved to a stream using the SaveToStream method. Use the PercentDone parameter to display progress information in your application while the table is being saved to the stream.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMTable.OnUpgradeLog Event

```
__property TLogEvent OnUpgradeLog
```

The UpgradeLog event is fired when a table is upgraded from an old table format using the UpgradeTable method. Use the LogMesssage parameter to display upgrade log information in your application while the table is being upgraded or to save the log messages to a file for later viewing.

TDBISAMTable.OnUpgradeProgress Event

```
__property TSteppedProgressEvent OnUpgradeProgress
```

The OnUpgradeProgress event is fired when a table is upgraded from an old table format using the UpgradeTable method. Use the Step and PercentDone parameters to display progress information in your application while the table is being upgraded.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

TDBISAMTable.OnVerifyLog Event

```
__property TLogEvent OnVerifyLog
```

The OnVerifyLog event is fired when a table is verified using the VerifyTable method and DBISAM needs to indicate the current status of the verification (such as the start or finish) or an error is found in the integrity of the table. Use the LogMesssage parameter to display verification log information in your application while the table is being verified or to save the log messages to a file for later viewing.

TDBISAMTable.OnVerifyProgress Event

```
__property TSteppedProgressEvent OnVerifyProgress
```

Occurs when a table is verified using the VerifyTable method. Use the Step and PercentDone parameters to display progress information in your application while the table is being repaired.

Note
The number of times that this event is fired is controlled by the TDBISAMSession ProgressSteps property.

5.25 TDBISAMUpdateSQL Component

Header File: dbisamtb

Inherits From TDBISAMSQLUpdateObject

Use the TDBISAMUpdateSQL component to update single or multiple source tables during the application of updates from a TClientDataSet component through the IProvider support in DBISAM. Usually the TDBISAMUpdateSQL component is used to handle complex updates to multiple tables that cannot be handled by the default IProvider support.

Properties	Methods	Events
DataSet	Apply	
DeleteSQL	ExecSQL	
InsertSQL	SetParams	
ModifySQL	TDBISAMUpdateSQL	
Query		
SQL		

TDBISAMUpdateSQL.DataSet Property

```
__property TDBISAMDataSet* DataSet
```

This property is automatically internally set by DBISAM.

TDBISAMUpdateSQL.DeleteSQL Property

```
__property System::Classes::TStrings* DeleteSQL
```

Use the DeleteSQL property to specify the DELETE statement to use when applying a deletion to a source table. Use parameters with the same names as any field names in the source table for any WHERE clause conditions, and use the prefix "OLD_" on any parameter names where you want an original field value to be used instead of the current field value being used for the update.

TDBISAMUpdateSQL.InsertSQL Property

```
__property System::Classes::TStrings* InsertSQL
```

Use the InsertSQL property to specify the INSERT statement to use when applying an insert to a source table. Use parameters with the same names as any field names in the source table.

TDBISAMUpdateSQL.ModifySQL Property

```
__property System::Classes::TStrings* ModifySQL
```

Use the ModifySQL property to specify the UPDATE statement to use when applying an update to a source table. Use parameters with the same names as any field names in the source table for any SET operations or WHERE clause conditions, and use the prefix "OLD_" on any parameter names where you want an original field value to be used instead of the current field value being used for the update.

TDBISAMUpdateSQL.Query Property

```
__property TDBISAMQuery* Query[Data::Db::TUpdateKind UpdateKind]
```

The Query property provides a reference to the internal TDBISAMQuery component actually used to execute the SQL in the InsertSQL, ModifySQL, and DeleteSQL properties.

TDBISAMUpdateSQL.SQL Property

```
__property System::Classes::TStrings* SQL[Data::Db::TUpdateKind UpdateKind]
```

The SQL property indicates the SQL statement in the InsertSQL, ModifySQL, or DeleteSQL property, depending on the setting of the UpdateKind index.

TDBISAMUpdateSQL.Apply Method

```
virtual void __fastcall Apply(Data::Db::TUpdateKind UpdateKind)
```

Call the Apply method to set the parameters for an SQL statement and execute it in order to update a record. The UpdateKind parameter indicates which SQL statement to bind and execute. The Apply method is primarily intended for manually executing update statements from an OnUpdateRecord event handler.

Note
If an SQL statement does not contain parameters, it is more efficient to call the ExecSQL method instead of the Apply method.

TDBISAMUpdateSQL.ExecSQL Method

```
void __fastcall ExecSQL(Data::Db::TUpdateKind UpdateKind)
```

Call the ExecSQL method to execute an SQL statement in order to update a record. The UpdateKind parameter indicates which SQL statement to execute.

Note
If the statement to execute contains any parameters, an application must call the SetParams method to bind the parameters before calling the ExecSQL method.

TDBISAMUpdateSQL.SetParams Method

```
void __fastcall SetParams(Data::Db::TUpdateKind UpdateKind)
```

Call the SetParams method to bind any parameters in an SQL statement associated with the update object prior to executing the statement. 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 source table.

Note
Parameter names can be prefaced by the "OLD_" prefix. If so, the old value of the field is used to perform the update instead of any updates in the cache.

TDBISAMUpdateSQL.TDBISAMUpdateSQL Method

```
__fastcall virtual TDBISAMUpdateSQL(System::Classes::TComponent*
      AOwner)
```

Use the New operator to create an instance of a TDBISAMUpdateSQL component using the constructor.

This page intentionally left blank

Chapter 6
Type Reference

6.1 TAbortErrorEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TAbortErrorEvent)(System::TObject* Sender,
      System::Sysutils::Exception* E, TAbortAction &Action)
```

This type is used for the TDBISAMQuery OnQueryError event.

6.2 TAbortProgressEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TAbortProgressEvent)(System::TObject* Sender, System::Word
      PercentDone, bool &Abort)
```

This type is used for the TDBISAMQuery OnQueryProgress event.

6.3 TCachedUpdateErrorEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TCachedUpdateErrorEvent)(System::TObject* Sender,
      TDBISAMRecord* CurrentRecord, System::Sysutils::Exception* E,
      TUpdateType UpdateType, Data::Db::TUpdateAction &Action)
```

This type is used for the TDBISAMTable and TDBISAMQuery OnCachedUpdateError event.

6.4 TCompressEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TCompressEvent)(System::TObject* Sender, const void * InBuffer,
      int InBytes, System::Byte Level, void * &OutBuffer, int
      &OutBytes)
```

This type is used for the TDBISAMEngine OnCompress event. Please see the Customizing the Engine topic for more information.

6.5 TCryptoInitEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TCryptoInitEvent)(System::TObject* Sender, void * Key, int
      KeyLen, void * &OutData, int &OutDataBytes)
```

This type is used for the TDBISAMEngine OnCryptoInit event. Please see the Customizing the Engine topic for more information.

6.6 TCryptoResetEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TCryptoResetEvent)(System::TObject* Sender, void * Data)
```

This type is used for the TDBISAMEngine OnCryptoReset event. Please see the Customizing the Engine topic for more information.

6.7 TCustomFunctionEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TCustomFunctionEvent)(System::TObject* Sender, const
      System::UnicodeString FunctionName, TDBISAMParams*
      FunctionParams, System::Variant &Result)
```

This type is used for the TDBISAMEngine OnCustomFunction event. Please see the Customizing the Engine topic for more information.

6.8 TDatabaseRights Type

Header File: dbisamtb

```
typedef System::Set<TDatabaseRight, TDatabaseRight::drRead,
      TDatabaseRight::drRestore> TDatabaseRights
```

This type is used as a set of TDatabaseRight enumerated types to indicate the set of rights currently granted to a given user for a given database on a database server. It is used as a parameter to the TDBISAMSession AddRemoteDatabaseUser, ModifyRemoteDatabaseUser, and GetRemoteDatabaseUser methods, as well as the TDBISAMEngine AddServerDatabaseUser, ModifyServerDatabaseUser, and GetServerDatabaseUser methods.

6.9 TDataLostEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TDataLostEvent)(System::TObject* Sender, TDataLossCause Cause,
      const System::UnicodeString ContextInfo, bool &Continue, bool
      &StopAsking)
```

This type is used for the TDBISAMTable and TDBISAMQuery OnDataLost event.

6.10 TDecompressEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TDecompressEvent)(System::TObject* Sender, const void *
      InBuffer, int InBytes, void * &OutBuffer, int &OutBytes)
```

This type is used for the TDBISAMEngine OnDecompress event. Please see the Customizing the Engine topic for more information.

6.11 TDecryptBlockEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TDecryptBlockEvent)(System::TObject* Sender, void * Data, void
      * BlockBuffer)
```

This type is used for the TDBISAMEngine OnDecryptBlock event. Please see the Customizing the Engine topic for more information.

6.12 TEncryptBlockEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TEncryptBlockEvent)(System::TObject* Sender, void * Data, void
      * BlockBuffer)
```

This type is used for the TDBISAMEngine OnEncryptBlock event. Please see the Customizing the Engine topic for more information.

6.13 TEndTransactionTriggerEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TEndTransactionTriggerEvent)(System::TObject* Sender,
      TDBISAMSession* TriggerSession, TDBISAMDatabase*
      TriggerDatabase)
```

This type is used for the TDBISAMEngine CommitTrigger and RollbackTrigger events.

6.14 TErrorEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TErrorEvent)(System::TObject* Sender, TDBISAMSession*
      ErrorSession, TDBISAMDatabase* ErrorDatabase, const
      System::UnicodeString TableName, TDBISAMRecord* CurrentRecord,
      System::Sysutils::Exception* E, Data::Db::TDataAction &Action)
```

This type is used for the TDBISAMEngine OnInsertError, OnUpdateError, OnDeleteError events.

6.15 TEventDays Type

Header File: dbisamtb

```
typedef System::StaticArray<bool, 7> TEventDays
```

This type is used to specify which days of the week an event should run on, with day 1 being Sunday and day 7 being Saturday. It is used a parameter to the TDBISAMSession AddRemoteEvent, ModifyRemoteEvent, and GetRemoteEvent methods, as well as the TDBISAMEngine AddServerEvent, ModifyServerEvent, and GetServerEvent methods.

6.16 TEventMonths Type

Header File: dbisamtb

```
typedef System::StaticArray<bool, 12> TEventMonths
```

This type is used to specify which months of the year an event should run on, with month 1 being January and month 12 being December. It is used a parameter to the TDBISAMSession AddRemoteEvent, ModifyRemoteEvent, and GetRemoteEvent methods, as well as the TDBISAMEngine AddServerEvent, ModifyServerEvent, and GetServerEvent methods.

6.17 TLogEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure *TLogEvent)(System::TObject*
      Sender, const System::UnicodeString LogMessage)
```

This type is used for the TDBISAMEngine OnServerLogEvent event.

6.18 TLoginEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TLoginEvent)(System::TObject* Sender, System::UnicodeString
      &UserName, System::UnicodeString &Password, bool &Continue)
```

This type is used for the TDBISAMSession OnRemoteLogin event.

6.19 TPasswordEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TPasswordEvent)(System::TObject* Sender, bool &Continue)
```

This type is used for the TDBISAMSession OnPassword event.

6.20 TProcedureProgressEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TProcedureProgressEvent)(System::TObject* Sender, const
      System::UnicodeString Status, System::Word PercentDone, bool
      &Abort)
```

This type is used for the TDBISAMSession OnRemoteProcedureProgress event.

6.21 TProcedureRights Type

Header File: dbisamtb

```
typedef System::Set<TProcedureRight, TProcedureRight::prExecute,
      TProcedureRight::prExecute> TProcedureRights
```

This type is used as a set of TProcedureRight enumerated types to indicate the set of rights currently granted to a given user for a given server-side procedure on a database server. It is used as a parameter to the TDBISAMSession AddRemoteProcedureUser, ModifyRemoteProcedureUser, and GetRemoteProcedureUser methods, as well as the TDBISAMEngine AddServerProcedureUser, ModifyServerProcedureUser, and GetServerProcedureUser methods.

6.22 TProgressEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TProgressEvent)(System::TObject* Sender, System::Word
      PercentDone)
```

This type is used for the TDBISAMQuery and TDBISAMTable OnIndexProgress, OnCopyProgress, OnOptimizeProgress, OnAlterProgress, OnImportProgress, OnExportProgress, OnLoadFromStreamProgress , OnSaveToStreamProgress, and OnSaveProgress events.

6.23 TReconnectEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TReconnectEvent)(System::TObject* Sender, bool &Continue, bool
      &StopAsking)
```

This type is used for the TDBISAMSession OnRemoteReconnect event.

6.24 TRecordLockTriggerEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TRecordLockTriggerEvent)(System::TObject* Sender,
      TDBISAMSession* TriggerSession, TDBISAMDatabase* TriggerDatabase,
       const System::UnicodeString TableName, int RecordNumber)
```

6.25 TSendReceiveProgressEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TSendReceiveProgressEvent)(System::TObject* Sender, int
      NumBytes, System::Word PercentDone)
```

This type is used for the TDBISAMSession OnRemoteSendProgress and OnRemoteReceiveProgress events.

6.26 TServerConnectEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TServerConnectEvent)(System::TObject* Sender, bool IsEncrypted,
      const System::UnicodeString ConnectAddress, System::TObject*
      &UserData)
```

This type is used for the TDBISAMEngine OnServerConnect event. Please see the Customizing the Engine topic for more information.

6.27 TServerDisconnectEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TServerDisconnectEvent)(System::TObject* Sender,
      System::TObject* UserData, const System::UnicodeString
      LastConnectAddress)
```

This type is used for the TDBISAMEngine OnServerDisconnect event. Please see the Customizing the Engine topic for more information.

6.28 TServerLogCountEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TServerLogCountEvent)(System::TObject* Sender, int &LogCount)
```

This type is used for the TDBISAMEngine OnServerLogCount event. Please see the Customizing the Engine topic for more information.

6.29 TServerLogEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TServerLogEvent)(System::TObject* Sender, const TLogRecord
      &LogRecord)
```

This type is used for the TDBISAMEngine OnServerLogEvent event. Please see the Customizing the Engine topic for more information.

6.30 TServerLoginEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TServerLoginEvent)(System::TObject* Sender, const
      System::UnicodeString UserName, System::TObject* UserData)
```

This type is used for the TDBISAMEngine OnServerLogin event. Please see the Customizing the Engine topic for more information.

6.31 TServerLogoutEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TServerLogoutEvent)(System::TObject* Sender, System::TObject*
      &UserData)
```

This type is used for the TDBISAMEngine OnServerLogout event. Please see the Customizing the Engine topic for more information.

6.32 TServerLogRecordEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TServerLogRecordEvent)(System::TObject* Sender, int Number,
      TLogRecord &LogRecord)
```

This type is used for the TDBISAMEngine OnServerLogRecord event. Please see the Customizing the Engine topic for more information.

6.33 TServerProcedureEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TServerProcedureEvent)(System::TObject* Sender, TDBISAMSession*
      ServerSession, const System::UnicodeString ProcedureName)
```

This type is used for the TDBISAMEngine OnServerProcedure event. Please see the Customizing the Engine topic for more information.

6.34 TServerReconnectEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TServerReconnectEvent)(System::TObject* Sender, bool
      IsEncrypted, const System::UnicodeString ConnectAddress,
      System::TObject* UserData)
```

This type is used for the TDBISAMEngine OnServerReconnect event. Please see the Customizing the Engine topic for more information.

6.35 TServerScheduledEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TServerScheduledEvent)(System::TObject* Sender, const
      System::UnicodeString EventName, bool &Completed)
```

This type is used for the TDBISAMEngine OnServerScheduledEvent event. Please see the Customizing the Engine topic for more information.

6.36 TSQLTriggerEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TSQLTriggerEvent)(System::TObject* Sender, TDBISAMSession*
      TriggerSession, TDBISAMDatabase* TriggerDatabase,
      TSQLStatementType StatementType, const System::UnicodeString SQL,
       double ExecutionTime, int RowsAffected)
```

This type is used for the TDBISAMEngine SQLTrigger event.

6.37 TStartTransactionTriggerEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TStartTransactionTriggerEvent)(System::TObject* Sender,
      TDBISAMSession* TriggerSession, TDBISAMDatabase* TriggerDatabase,
       System::Classes::TStrings* Tables)
```

This type is used for the TDBISAMEngine StartTransactionTrigger event.

6.38 TSteppedProgressEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TSteppedProgressEvent)(System::TObject* Sender, const
      System::UnicodeString Step, System::Word PercentDone)
```

This type is used for the TDBISAMQuery and TDBISAMTable OnVerifyProgress, OnRepairProgress, and OnUpgradeProgress events.

6.39 TTextIndexFilterEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TTextIndexFilterEvent)(System::TObject* Sender, const
      System::UnicodeString TableName, const System::UnicodeString
      FieldName, System::UnicodeString &TextToIndex)
```

This type is used for the TDBISAMEngine OnTextIndexFilter event. Please see the Customizing the Engine topic for more information.

6.40 TTextIndexTokenFilterEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TTextIndexTokenFilterEvent)(System::TObject* Sender, const
      System::UnicodeString TableName, const System::UnicodeString
      FieldName, const System::UnicodeString TextIndexToken, bool
      &Include)
```

This type is used for the TDBISAMEngine OnTextIndexTokenFilter event. Please see the Customizing the Engine topic for more information.

6.41 TTimeoutEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TTimeoutEvent)(System::TObject* Sender, bool &StayConnected)
```

This type is used for the TDBISAMSession OnRemoteTimeout event.

6.42 TTraceEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TTraceEvent)(System::TObject* Sender, const TTraceRecord
      &TraceRecord)
```

This type is used for the TDBISAMSession OnRemoteTrace event.

6.43 TTriggerEvent Type

Header File: dbisamtb

```
typedef void __fastcall (__closure
      *TTriggerEvent)(System::TObject* Sender, TDBISAMSession*
      TriggerSession, TDBISAMDatabase* TriggerDatabase, const
      System::UnicodeString TableName, TDBISAMRecord* CurrentRecord)
```

This type is used for the TDBISAMEngine BeforeInsertTrigger, AfterInsertTrigger, BeforeUpdateTrigger, AfterUpdateTrigger, BeforeDeleteTrigger, or AfterDeleteTrigger events.

This page intentionally left blank

Appendix A - Differences from the BDE

There are several key differences between the BDE and DBISAM that should be taken into account, especially when converting an existing application that uses the BDE over to DBISAM.

> **Note**
> All comparisons below assume usage of only the Paradox and dBase/FoxPro local table formats available in the BDE. For numerous reasons Access is not included in this comparison.

Difference	Further Details
BLOBs	The BDE allows (using Paradox tables) a portion of BLOB fields to be stored in the actual records in addition to storing the BLOB fields in a separate physical BLOB file. DBISAM does not support this and more closely resembles the dBase table format where all of the BLOB fields are stored in a separate physical BLOB file. Also, DBISAM always buffers BLOBs in memory when records are being added or edited. This is in contrast to the BDE, which will write overflow BLOB data to temporary files on disk when they exceed an acceptable amount of memory consumption (acceptable is determined internally by the BDE).
Batch Moves	DBISAM does not currently contain a batch move component, however bulk inserts and updates can be accomplished via SQL INSERT and UPDATE statements. DBISAM also allows for importing and exporting tables to text files via the ExportTable and ImportTable methods of the TDBISAMTable and TDBISAMQuery components.
Cached Updates	DBISAM supports cached updates, although there are some differences between the way the BDE handles cached updates and DBISAM handles cached updates. DBISAM caches the entire source dataset immediately upon beginning cached updates, whereas the BDE caches records "on demand" as the dataset is updated. Also, the reconciliation event is different for DBISAM, although the basic principles of reconciliation are the same.
Concurrency (Multi-User Usage)	The BDE in general, and Paradox tables in particular, can be difficult to set up for proper multi-user usage. There is the .NET file, .LCK files, and the Local Share setting to deal with and improper settings can cause data refresh problems and in the worst case, data corruption and loss of data. With DBISAM, all locking is done through the operating system and does not involve any external files that must be configured. There is absolutely nothing extra that you must do when writing an application for single-user or multi-user use. All of the hard work is done for you, transparently, and the only task left up to

	you is copying the tables onto the network file server. Please see the Locking and Concurrency topic for more information.
Distribution	DBISAM can be compiled completely into your application and does not require any other DLLs or external components. Runtime package support is also provided with the dbxxxxxr.bpl (Delphi and C++Builder) or bpldbxxxxx.so (Kylix) package that is distributed with DBISAM (the XXXXX is replaced with a 3-digit DBISAM version number, for example 300 is version 3.00, and a d for Delphi, a c for C++Builder, or a k for Kylix, along with a single digit version number such as 5 for Delphi 5).
Encryption and User Security	The BDE includes support for user security through encryption and passwords with the Paradox table format. DBISAM also supports encrypting a table with a password as well. Whenever you attempt any operation on an encrypted table in DBISAM you will be prompted for the password. You should be extremely careful with this functionality since you will not be able to open an encrypted table if you lose the password. The encryption in DBISAM is a strong 8-byte block cipher called Blowfish and can be replaced with the block cipher of your choice. Please see the Starting Sessions and Opening Tables topics for more information. In addition to this basic security, the DBISAM database server allows for a more complete security model by offering complete user-based security along with database-level rights that can be assigned on a per-user basis. Also, all communications in the with the DBISAM database server can be encrypted with the same strong Blowfish encryption technology to prevent any data from being "sniffed" on the network. By default, all administrative access and all login and password information are automatically encrypted and are never sent in an unencrypted fashion over any network. Please see the Encryption, Server Administration, and Customizing the Engine topics for more information.
Error Trapping	DBISAM uses its own exception class called EDBISAMEngineError, not the BDE-specific EDBEngineError exception class. The behavior between the the two exception classes is similar, however the EDBISAMEngineError exception class only contains an ErrorCode property (and some additional context information) whereas the EDBEngineError exception class contains an array of error classes, each with its own error code. The reason for this is that the BDE can raise multiple error codes in one exception, whereas DBISAM only raises one error per exception. Please see the Exception Handling and Errors topic for more information.
Filters	Both the BDE and DBISAM provide expression filters through the Filter, FilterOptions, and Filtered properties,

	and a callback-based filter mechanism exposed as the OnFilterRecord event that allows you to filter on any type of arbitrary data. DBISAM uses straight SQL syntax for its filters, including the ability to use SQL functions and extended operators such as IN, LIKE, and BETWEEN. Because of this, filters in DBISAM are strongly-typed and you cannot mix strings with integers, and vice-versa. The BDE uses a pseudo-SQL filter syntax that allows some implicit type conversions such as this. However, other than this difference the two syntaxes are almost exactly the same for basic expressions that do not use SQL-specific extensions. The expression filters in both DBISAM and the BDE are optimized by the database engine, which means that whenever an index (either primary or secondary) is available that satisifies a portion or all of the filter criteria, then that index is used to locate the appropriate records instead of scanning top to bottom through the actual data records. Please see the Filter Optimization topic for more information. The record sequencing when filters are in effect is identical to that of the BDE as well as the record counts. One of the main differences between the BDE and DBISAM is the inclusion of NULL values when performing less than (or equal to) comparisons. The BDE will include NULLs in the result of such a filter, but DBISAM will not. The standard behavior for SQL selection is to not include NULLs, which is the reason behind this difference. Please see the Setting Filters on Tables topic for information.
Free Space Management	The BDE practices free space recycling in the Paradox table format but not in the dBase or FoxPro table formats. In these formats free space is not recycled and after many inserts and deletes the tables tend to get bloated. This is primarily because with these formats the records are not physically deleted, only marked for deletion, and their corresponding index keys are kept in the indexes. DBISAM recycles all free space in data records, indexes, and BLOBs transparently and without any user intervention required. When a record is deleted, the space is marked as free and available for re-use immediately. An OptimizeTable method is also provided with the TDBISAMTable component that allows you to optimize a table for a particular index order (i.e. "clustering" the table), optimize BLOB access, and also remove any free space from the data records, indexes, and BLOBs. Please see the Optimizing Tables topic for more information.
In-Memory Tables	The BDE allows for the creation of in-memory tables that behave somewhat like a regular table, but are severely

	limited in several areas such as the ability to add and use indexes (both primary and secondary) and the ability to use BLOB fields. Also, an in-memory table created through the BDE is automatically destroyed upon closing and cannot be shared by multiple TTable components. The low-level BDE calls to take advantage of in-memory tables are also quite confusing to most novice developers. DBISAM, on the other hand, overcomes all of these limitations and even allows in-memory tables to be shared by multiple TDBISAMTable components. Also, to use in-memory tables in DBISAM is as simple as setting the DatabaseName property to "Memory" for the TDBISAMTable component. Please see the In-Memory Tables and Opening Tables topics for more information. **Note** Because in-memory tables in DBISAM act like regular disk-based tables, you must first create the table using the TDBISAMTable CreateTable method and delete the table using the TDBISAMTable DeleteTable method to get rid of the table. You can also use the SQL CREATE TABLE and DROP TABLE statements to perform the equivalent functions using only SQL.
Indexes	The BDE supports several different indexing schemes for local databases through separate drivers. DBISAM most closely resembles the Paradox table format in that it supports primary and secondary indexes, but it does offer some features that are found in the FoxPro and dBase index formats also. The following are notes on the differences between the BDE local database index formats and DBISAM's index format: • Primary Indexes Paradox allows for tables with no primary index defined. DBISAM also supports tables without a primary index defined by the user or developer, but will automatically define a primary index if one is not defined explicitly. This automatically defined primary index is based upon the special, non-changing, RecordID psuedo-field found in every DBISAM record. • Case-Insensitivity Paradox supports case insensitive indexes for secondary indexes. DBISAM also support case-insensitive indexes, but for both primary and secondary indexes. • Secondary Indexes Paradox stores secondary index definitions in separate files, one file for each separate index. This is in addition

to a separate file for the primary index. DBISAM stores all primary and secondary indexes in one .idx file, which cuts down on file handle usage and headaches associated with distribution. This is similar to the index formats of FoxPro and dBase.

• Key Compression

The Paradox and dBase index formats do not include any type of index key compression at all, while the FoxPro index format includes very good index key compression. DBISAM allows you to choose from duplicate-byte compression, trailing-byte compression, or full compression (both types of compression combined). The Foxpro index format always uses full compression and does not allow you to choose the compression method.

• Descending Indexes

The Paradox index format allows for descending secondary indexes in version 7.0 and above. DBISAM allows for descending indexes also, but for both primary and secondary indexes. Both the Paradox and DBISAM index formats allow for individual fields within an index to be marked as descending so that you an mix ascending with descending fields in the same index.

In addition to the index formats, the BDE supports several features for searching on indexes and setting ranges that are noteworthy:

• Partial Field Search

The BDE will allow you to search on a partial field count of the entire active index. For example, if you had a primary index on a Paradox table that consisted of CustomerNum, OrderNum, and LineNum, the BDE would allow you to search on just CustomerNum to find the record you are looking for. DBISAM also supports this feature in the exact same manner as the BDE.

• Partial Field Range

The BDE also allows you to use this same principle when applying ranges to a table, and again DBISAM fully supports this method of setting ranges.

• Logical Record #'s

When using the Paradox table format with the BDE, you can retrieve a logical record number based upon the currently active index. DBISAM supports this feature.

• Exact Record Count

Paradox allows you to get an instantaneous and exact

	record count even when a range is currently set. DBISAM also supports this feature.
International Support	Both the BDE and DBISAM provide international support in the form of proper collation and sorting of indexes for tables that use a non-US language. DBISAM provides international support for Delphi and C++Builder through the Windows locale support provided in the operating system. The locale can be specified on a table basis and multiple tables with multiple locales can be used in the same application. For Kylix applications, the only locale supported currently is the default ANSI Standard locale provided by DBISAM. DBISAM will raise an exception if a table is attempted to be opened or created on a machine that does not have the proper support installed in the operating system for a desired locale, including Linux. Please see the Creating and Altering Tables and Opening Tables topics for more information. In contrast, the international support in the BDE is provided via custom language drivers that are provided by Borland with the BDE. **Note** DBISAM does not support international date, time, or number formats in filter or SQL statements, but rather uses a standard ANSI format for all dates, times, and numbers.
Low-Level API Calls	The BDE requires that you must use low-level API calls to accomplish certain tasks. Restructuring tables, providing progress information to users during batch processes, and copying tables require lengthy and cryptic amounts of code to accomplish the desired task. DBISAM provides well-documented and easy-to-use properties, methods, and events via the various DBISAM components for these purposes and completely removes the need to make any API calls at all.
Memory Usage	DBISAM does not pre-allocate memory for data, index, and BLOB buffers like the BDE does. In contrast, DBISAM only allocates memory for caching on an as-needed basis and is restricted to the following limits by default: • Data Record Buffers 32 kilobytes is the maximum amount of record buffer cache and 8192 record buffers are the maximum that will be cached at one time • Index Page Buffers 64 kilobytes is the maximum amount of index page buffer cache and 8192 index page buffers are the maximum that will be cached at one time

	• BLOB Block Buffers 32 kilobytes is the maximum amount of BLOB block buffer cache and 8192 BLOB block buffers are the maximum that will be cached at one time **Note** These figures are on a per physical table basis within the same session. Tables opened in different sessions use different caches, hence they use additional memory. In contrast, if you open the same physical table multiple times within the same session DBISAM will only use one cache for the table and share this cache among all of the open table instances. Also, if the number of records updated in a transaction exceeds the above figures then the memory consumption will automatically be expanded to accomodate the transacton and reset back to the default values when the transaction commits or rolls back. These figures can be changed via the following properties in the TDBISAMEngine component, so they are only default values: • MaxTableDataBufferCount property • MaxTableDataBufferSize property • MaxTableIndexBufferCount property • MaxTableIndexBufferSize property • MaxTableBlobBufferCount property • MaxTableBlobBufferSize property
NULL Support	DBISAM includes complete NULL support that behaves identically to that of Paradox, although it is implemented in a safer and more thorough manner. The rules for NULL support in DBISAM are as follows: • If a field has not been assigned a value and was not defined as having a default value in the table structure, it is NULL. • As soon as a field has been assigned a value it is not considered NULL anymore. String, FixedChar, GUID, Blob, Memo, and Graphic fields are an exception this rule. When you assign a NULL value (empty string) to a String, FixedChar, or GUID field the field will be set to NULL. When the contents of a Blob, Memo, or Graphic field are empty, i.e. the length of the data is 0, the field will be set to NULL. • If the Clear method of a TField object is called the field will be set to NULL.

	• NULL values are treated as separate, distinct values when used as an index key. For example, let's say that you have a primary index comprised of one Integer field. If you had a field value of 0 for this Integer field in one record and a NULL value for this Integer field in another record, DBISAM will not report a key violation error. This is a very important point and should be considered when designing your tables. As a general rule of thumb, you should always provide values for fields that are part of the primary index. • The BDE will include NULL values when performing less than (or equal to) comparisons in filters or SQL queries, but DBISAM will not. The standard behavior for SQL selection is to not include NULLs, which is the behavior that DBISAM uses.
Performance	DBISAM handles automatic change detection differently than the BDE, and this can cause differences in performance. With the Paradox table format, the BDE only checks for changes on disk (and subsequently refreshes its local buffers) when it needs to read data physically from the disk and when also when a record lock is acquired. With the dBase and FoxPro table formats, the BDE will never refresh local buffers unless it is forced to read data physically from the disk or a record lock is acquired. By default DBISAM uses the same type of change detection as the BDE does with the Paradox table format. This is controlled by the StrictChangeDetection property of the TDBISAMSession component. The default value is False, indicating that lazy change detection is in effect. If this property is set to True, indicating that strict change detection is in effect, DBISAM will always check for changes by other users before any read operation and will always ensure that its local buffers contain the most up-to-date data. Please see the Change Detection topic for more information. Finally, the BDE has a Local Share setting that determines whether writes are cached for shared tables. Setting the Local Share setting to False can lead to very disastrous results if the application is unexpectedly terminated since it can cause all updates to be lost forever. DBISAM, on the other hand, never caches writes for shared tables. If you wish to ensure that the operating system subsequently physically flushes the updates to disk (some operating systems such as Windows 95/98/ME/NT/2000/XP may cache writes for a short period of time) you may use the TDBISAMTable or TDBISAMQuery FlushBuffers method or the TDBISAMSession ForceBufferFlush property to do so. Please see the Buffering and Caching topic fore more information.

	Note Opening a table exclusively in DBISAM will cause DBISAM to cache all writes, which will result in excellent performance. However, an unexpected termination of the application can cause data loss similar to setting Local Share to False with the BDE. Please see the Opening Tables topic for more information.
Queries (SQL and QBE)	DBISAM includes complete support for queries using SQL SELECT, INSERT, UPDATE, DELETE, CREATE TABLE, ALTER TABLE, DROP TABLE, CREATE INDEX, DROP INDEX statements that is almost entirely compliant with the Local SQL syntax available in the BDE. The features that are currently not supported in DBISAM's implementation of these SQL statements include: • FULL OUTER JOIN clause DBISAM does not provide for a FULL OUTER JOIN clause, although LEFT OUTER JOINs and RIGHT OUTER JOINs are fully supported. • ANY or EXISTS operators for sub-selects DBISAM does not provide for the ANY or EXISTS operators for specifying sub-select predicates in WHERE clauses. However, DBISAM does allow for using the IN operator for sub-select queries within an SQL SELECT statement. • DISTINCT clause with aggregates DBISAM does not support the use of the DISTINCT clause with the COUNT, MIN, MAX, AVG, SUM, or RUNSUM aggregate functions. DBISAM also includes many additional features and enhancements that are not found in the Local SQL syntax provided by the BDE. Please see the SQL Reference Overview topic for more information. DBISAM also does not support using QBE for queries. The BDE issues cryptic SQL error messages that usually provide little or no indication of where an error exists in an SQL statement. DBISAM always provides for complete and descriptive error messages that indicate exactly where an error exists in the SQL syntax. DBISAM supports the use of multi-statement SQL scripts in the TDBISAMQuery component, complete with parameter events for each statement, progress events, and error-trapping events. DBISAM SQL statements such as ALTER TABLE and CREATE INDEX fire progress and

	status events through the TDBISAMQuery component. Please see the Executing SQL Queries topic for more information.
Stored Procedures	DBISAM does not support stored procedures in the traditional sense or per the SQL standards. However, DBISAM does provide for server-side procedures implemented via the TDBISAMEngine::OnServerProcedure event in the TDBISAMEngine component, and this event can very easily launch any type of scripting language on the server to implement a scripted server-side procedure similar to a traditional stored procedure. You can also use the TDBISAMEngine OnServerScheduledEvent event to implement scheduled processes that run on the server in the same fashion as a server-side procedure. Please see the Customizing the Engine topic for more information. Also, DBISAM provides support for DDL and DML SQL statements in scripts within the SQL property of the TDBISAMQuery component. This allows you to write scripts that contain multiple SQL statements. The only requirement is that each SQL statement is separated by a semicolon. Please see the Executing SQL Queries topic for more information.
Transaction Support	Transaction support in the BDE for local databases is somewhat limited in the amount of records that can participate in a given transaction. The highest limit currently imposed is 255 records and this depends upon whether you're using the dBase/Foxpro or Paradox table formats. DBISAM does not impose any limits on transactions, and you may have as many records participating in a transaction as available memory will allow. Unlike the BDE, which uses a log-based transaction system for local database formats, DBISAM implements transactions completely in memory and buffers all updates during a transaction. These updates are not written to disk until the transaction is committed, and are discarded if the transaction is rolled back. Also, the BDE uses a dirty-read transaction isolation level for transactions on local databases, whereas DBISAM always uses a read-committed transaction isolation level for all transactions. Please see the Transactions topic for more information.

	Note Neither the BDE or DBISAM offer fail-safe transactions, so do not rely on them to prevent data inconsistencies with complicated transactions that affect multiple files unless you can guarantee that the application will not get unexpectedly interrupted. DBISAM is slightly more immune to these type of problems due to its buffered transaction design, but problems may still arise if the application is unexpectedly interrupted during the process of committing a transaction.

This page intentionally left blank

Appendix B - Error Codes and Messages

The following is a table of the error messages used by DBISAM, their corresponding error codes and constants used to represent them, and additional information about the error message. DBISAM uses the EDBISAMEngineError exception object to raise exceptions when an engine error occurs. In each error message below, the specific properties that will be populated in the EDBISAMEngineError object are indicated by the <> brackets. The ErrorCode property is always populated with the error code of the current exception. In addition, certain properties such as the ErrorDatabaseName, ErrorRemoteName (Client-Server only), and ErrorUserName (Client-Server only) properties are almost always populated even if they are not used in the actual error message.

Note
This list only covers the exceptions raised by the DBISAM database engine itself and does not cover the general EDatabaseError exceptions raised by the component units. If you wish to use the error constants defined by DBISAM in your applications for error trapping you need to make sure:

- For Delphi applications, that the dbisamcn unit file is included in your uses clause for the source unit in question

- For C++ applications, that the dbisamcn header file is included in your .h header file for the source file in question

If you wish to change the following error messages or translate them into a different language, you may do so by altering the contents of the dbisamst unit that can be found in the same directory where the other DBISAM units were installed. This file includes both the DBISAM database engine error messages shown below and the more general database errors raised by the component units as resource strings. After altering this unit be sure to recompile your application so that the changes will be incorporated into your application.

For more information on exception handling and error trapping in your application along with the exception types used by DBISAM, please see the Exception Handling and Errors topic in this manual.

Error Code	Message and Further Details
DBISAM_BOF (8705)	Beginning of table '<TableName>' unexpectedly encounteredThis error should not normally occur since encountering the beginning of a table is a normal occurrence. The only time this error is triggered is when the beginning of the table is encountered unexpectedly. If this error does occur the most likely reason is corruption in the table, and you should repair the table. Please see the Verifying and Repairing Tables topic for more information.
DBISAM_EOF (8706)	End of table '<TableName>' unexpectedly encounteredThis error should not normally occur since encountering the end of the table is a normal occurrence. The only time this error is triggered is when the end of the table is encountered unexpectedly. This error should never occur, however if it does the most likely reason is corruption in the table. You should repair the table if this error occurs. Please see the Verifying

	and Repairing Tables topic for more information.
DBISAM_KEYORRECDELETED (8708)	Record has been changed or deleted by another user, session, or table cursor in the table '<TableName>'This error occurs when an attempt is made to edit or delete a record and the record has already been changed or deleted by another user, session, or table cursor since DBISAM last cached the record. Please see the Updating Tables and Query Result Sets and Change Detection topics for more information.
DBISAM_NOCURRREC (8709)	No current record in the table '<TableName>'This error occurs when DBISAM internally attempts to retrieve the current record and the current record is not present, most likely due to the table being empty because of a range or filter condition. Please see the Setting Filters on Tables and Query Result Sets, Setting Ranges on Tables, and Searching and Sorting Tables and Query Result Sets topics for more information.
DBISAM_RECNOTFOUND (8710)	Record not found in the table '<TableName>'This error occurs when when a call to the TDBISAMTable GotoCurrent method fails to find the corresponding record in the destination table parameter.
DBISAM_ENDOFBLOB (8711)	End of BLOB field in the table '<TableName>' reached prematurelyThis error occurs when DBISAM attempts to read from a BLOB field that is expected to be x number of bytes in length but encounters the end of the BLOB before reaching the desired number of bytes. This error should never occur, however if it does the most likely reason is corruption in the table. You should repair the table if this error occurs. Please see the Verifying and Repairing Tables topic for more information.
DBISAM_HEADERCORRUPT (8961)	Header information corrupt in table <TableName>This error occurs when DBISAM opens a table and detects that the actual data in the table does not match the header information, which is always an indication of corruption in the table. Please see the Verifying and Repairing Tables topic for more information.
DBISAM_FILECORRUPT (8962)	Data record buffers corrupt in the table '<TableName>'This error occurs when DBISAM attempts to read, write, or manipulate an internal record buffer and cannot do so due to table corruption. You should repair the table if this error occurs. Please see the Verifying and Repairing Tables topic for more information.
DBISAM_MEMOCORRUPT (8963)	BLOB block buffers corrupt in the table '<TableName>'This error occurs when DBISAM attempts to read, write, or manipulate an internal BLOB block buffer and cannot do so due to table corruption. You should repair the table if this error occurs. Please see the Verifying and Repairing Tables topic for more information.
DBISAM_INDEXCORRUPT (8965)	Index page buffers corrupt in the table '<TableName>'This error occurs when DBISAM attempts

	to read, write, or manipulate an internal index page buffer and cannot do so due to table corruption. You should repair the table if this error occurs. Please see the Verifying and Repairing Tables topic for more information.
DBISAM_READERR (9217)	Error reading from the table or backup file '<Name>'This error occurs when DBISAM attempts to read from the table or backup file and cannot read the desired number of bytes. This error should never occur, however if it does the most likely reason is corruption in the table or backup file. You should repair the table if this error occurs with a table, but there is currently no way to recover data from a corrupted backup file. Please see the Verifying and Repairing Tables topic for more information on repairing tables.
DBISAM_WRITEERR (9218)	Error writing to the table or backup file '<Name>'This error occurs when DBISAM attempts to write to a table or backup file and cannot write the desired number of bytes. This error should never occur, however if it does the most likely reason is corruption in the table or backup file, or a lack of available disk space. You should repair the table if this error occurs, or if the error occurs for a backup file, check the available disk space and restart the backup process. Please see the Verifying and Repairing Tables topic for more information.
DBISAM_RECTOOBIG (9477)	Maximum record size exceeded in the table '<TableName>'This error occurs when an attempt is made to create a new table or alter an existing table's structure and doing so would exceed the maximum allowable record size. Please see Appendix C - System Capacities for more information.
DBISAM_TABLEFULL (9479)	The table '<TableName>' is full and cannot contain any more dataThis error occurs when an attempt is made to add data to a table and doing so would cause any one of the physical files that make up a logical table to be larger than the allowable maximum file size, or it would cause the table to exceed the maximum number of allowable records. Please see Appendix C - System Capacities and the Enabling Large File Support for more information.
DBISAM_DATABASEFULL (9480)	The database '<DatabaseName>' is full and cannot contain any more tablesThis error occurs when an attempt is made to create a new table and doing so would cause the number of tables to exceed the maximum number of tables allowed for a given database. Please see Appendix C - System Capacities and the Creating and Altering Tables for more information.
DBISAM_INTERNALLIMIT (9482)	Too many filters defined for the table '<TableName>'This error occurs when DBISAM internally attempts to define more filters than allowed. This error should never occur so if it does you should contact Elevate Software immediately for a resolution to the

	problem.
DBISAM_INDEXLIMIT (9487)	Maximum or minimum limits on the index page size, the number of indexes, or the number of fields in an index exceeded in the table '<TableName>'This error occurs when an attempt is made to add a new primary or secondary index, create a new table, or alter an existing table's structure and doing so would exceed the maximum or minimum index page size, the maximum number of indexes, or the maximum number of fields per index that can be present in a table. Please see Appendix C - System Capacities for more information.
DBISAM_FLDLIMIT (9492)	Maximum or minimum limits on number of fields exceeded in the table '<TableName>'This error occurs when an attempt is made to create a new table or alter an existing table's structure and doing so would exceed the maximum or minimum number of fields that can be present in a table. Please see Appendix C - System Capacities for more information.
DBISAM_OPENBLOBLIMIT (9494)	Too many BLOBs opened in the table '<TableName>'This error occurs when DBISAM attempts to open more BLOBs than what is allowed. Please see Appendix C - System Capacities for more information.
DBISAM_BLOBLIMIT (9498)	Too many BLOB fields or invalid BLOB block size specified for the table '<TableName>'This error occurs when DBISAM internally attempts to open more blobs than allowed or attempts to create a table or alter an existing table's structure with an invalid BLOB block size. Please see Appendix C - System Capacities for more information.
DBISAM_KEYVIOL (9729)	Duplicate key found in the index '<IndexName>' of the table '<TableName>'This is a non-fatal error that occurs when an attempt is made to add a record to a table and the record would cause a duplicate key to be added for the primary index or for a secondary index that it is defined as being unique. Primary indexes implicitly enforce uniqueness and therefore cannot permit duplicates to be added, while secondary indexes must be explicitly defined as unique. Please see the Updating Tables and Query Result Sets topic for more information.
DBISAM_MINVALERR (9730)	The value for the field '<FieldName>' in the table '<TableName>' is below the minimum allowable valueThis is a non-fatal error that occurs when an attempt is made to add a record to a table or update a record in a table, and a field in the record would violate a minimum value constraint for that field. Constraints ensure that data is either present or within an approved range of values for any given field. Please see the Updating Tables and Query Result Sets and Creating and Altering Tables topics for more information.
DBISAM_MAXVALERR (9731)	The value for the field '<FieldName>' in the table '<TableName>' is above the maximum allowable valueThis is a non-fatal error that occurs when an

	attempt is made to add a record to a table or update a record in a table, and a field in the record would violate a maximum value constraint for that field. Constraints ensure that data is either present or within an approved range of values for any given field. Please see the Updating Tables and Query Result Sets and Creating and Altering Tables topics for more information.
DBISAM_REQDERR (9732)	A value must be provided for the field '<FieldName>' in the table '<TableName>'This is a non-fatal error that occurs when an attempt is made to add a record to a table or update a record in a table, and a field in the record would violate a required constraint for that field. Constraints ensure that data is either present or within an approved range of values for any given field. Please see the Updating Tables and Query Result Sets and Creating and Altering Tables topics for more information.
DBISAM_OUTOFRANGE (9985)	Invalid field number or name '<FieldName>' specified for the table '<TableName>'This error occurs when a field number is referenced in an operation and that field number does not exist.
DBISAM_PRIMARYKEYREDEFINE (9993)	A primary index is already defined for the table '<TableName>' and cannot be added againThis error occurs when an attempt is made to add a primary index to a table when one already exists. You must first remove the existing primary index before you can add a new one. Please see the Adding and Deleting Indexes from a Table and Creating and Altering Tables topics for more information.
DBISAM_INVALIDBLOBOFFSET (9998)	Invalid BLOB offset into the table '<TableName>' specifiedThis error occurs when DBISAM attempts to access a BLOB field in a table and the offset at which it is trying to access the data is invalid. This error should never occur, however if it does the most likely reason is corruption in the table. You should repair the table if this error occurs. Please see the Verifying and Repairing Tables topic for more information.
DBISAM_INVALIDFLDTYPE (10000)	Invalid field definition specified for the field '<FieldName>' in the table '<TableName>'This error occurs when an attempt is made to create a new table or alter an existing table's structure, and a field definition specified is invalid due to an incorrect field number, data type, or length. Please see the Creating and Altering Tables topic for more information.
DBISAM_INVALIDVCHKSTRUCT (10006)	Invalid default expression or min/max constraint specified for the field '<FieldName>' in the table '<TableName>'This error occurs when a default value or minimum/maximum constraint expression for the indicated field is invalid. You need to verify that the expression provided is appropriate for the data type of the indicated field. If the expression contains constants, you need to verify that the expression is properly formatted, especially with date, time, or number expressions. Please see the Creating and Altering Tables

	topic for more information.
DBISAM_INDEXNAMEREQUIRED (10010)	The secondary index name is missing or not specified for the table '<TableName>'This error occurs when an attempt is made to add a new secondary index and the secondary index name is missing or not specified. All secondary indexes require a unique name (case-insensitive) that is used to identify the index. Please see the Adding and Deleting Indexes from a Table and Creating and Altering Tables topics for more information.
DBISAM_INVALIDPASSWORD (10015)	Password provided for the table '<TableName>' is invalidThis error occurs when an attempt is made to create a table or alter an existing table's structure with a password that is invalid. Please see the Creating and Altering Tables topic for more information.
DBISAM_INVALIDINDEXNAME (10022)	Invalid secondary index name '<IndexName>' specified for the table '<TableName>'This error occurs when an attempt is made to set the active index to a secondary index that does not exist, delete a secondary index that does not exist, create a table or alter an existing table's structure with an index name that is invalid, or add a secondary index with an invalid index name. Please see the Searching and Sorting Tables and Query Result Sets, Adding and Deleting Indexes from a Table, and Creating and Altering Tables topics for more information.
DBISAM_INVALIDIDXDESC (10023)	Invalid field number '<FieldName>' specified in the index '<IndexName>' for the table '<TableName>'This error occurs when an index definition contains an invalid field number. This error should never occur, however if it does the most likely reason is corruption in the table. You should repair the table if this error occurs. Please see the Verifying and Repairing Tables topic for more information.
DBISAM_INVALIDKEY (10026)	Invalid key size specified in the index '<IndexName>' for the table '<TableName>'This error occurs when an attempt is made to add a new primary or secondary index, create a new table, or alter an existing table's structure, and doing so would exceed the maximum length of an index key that can be present in a table. Please see Appendix C - System Capacities for more information.
DBISAM_INDEXEXISTS (10027)	The secondary index '<IndexName>' already exists for the table '<TableName>'This error occurs when an attempt is made to add a new secondary index and the secondary index name already exists in the table. All secondary indexes require a unique name (case-insensitive) that is used to identify the index. Please see the Adding and Deleting Indexes from a Table and Creating and Altering Tables topics for more information.
DBISAM_INVALIDBLOBLEN (10029)	Invalid BLOB length in the table '<TableName>'This error occurs when DBISAM attempts to access a BLOB field in the table and the length of the BLOB it is trying to access is invalid. This error should never occur,

	however if it does the most likely reason is corruption in the table. You should repair the table if this error occurs. Please see the Verifying and Repairing Tables topic for more information.
DBISAM_INVALIDBLOBHANDLE (10030)	Invalid BLOB handle for the table '<TableName>' specifiedThis error occurs when DBISAM internally attempts to access a BLOB field with an invalid handle to the BLOB. This error should never occur, however if it does the most likely reason is corruption in the table. You should repair the table if this error occurs. Please see the Verifying and Repairing Tables topic for more information.
DBISAM_TABLEOPEN (10031)	The table '<TableName>' is already in useThis error occurs when an attempt is made to perform an operation on a table that requires the table to be closed in order to complete the operation. The operations that require the table to be closed are as follows: • Verifying and Repairing Tables • Creating and Altering Tables • Optimizing Tables • Upgrading Tables • Deleting Tables • Renaming Tables
DBISAM_INVALIDFIELDNAME (10038)	Invalid or duplicate field name '<FieldName>' specified for the table '<TableName>'This error occurs when an attempt is made to create a table or alter an existing table's structure with a field name that is invalid. Please see the Creating and Altering Tables topic for more information.
DBISAM_NOSUCHFILTER (10050)	Invalid filter handle specified for the table '<TableName>'This error occurs when an attempt is made by DBISAM to refer to a filter expression or callback filter that does not exist. This error should never occur so if it does you should contact Elevate Software immediately for a resolution to the problem.
DBISAM_INVALIDFILTER (10051)	Filter error for the table '<TableName>' - <Message>This error occurs when an invalid filter statement is passed to DBISAM. The most common cause of this error is improperly formatted date, time, or number constants or some other type of syntax error. Please see the Setting Filters on Tables and Query Result Sets topic for more information.
DBISAM_LOCKREADLOCK (10221)	Cannot read lock the lock file for the database '<DatabaseName>'This error occurs when DBISAM internally attempts to place a read lock on the lock file (dbisam.lck) for a given database and it fails. Under normal circumstances this error should never occur so if

	it does you should contact Elevate Software immediately for a resolution to the problem. Please see the Locking and Concurrency topic for more information.
DBISAM_LOCKREADUNLOCK (10222)	Cannot read unlock the lock file for the database '<DatabaseName>'This error occurs when DBISAM internally attempts to remove a read lock on the lock file (dbisam.lck) for a given database and it fails. This error should never occur so if it does you should contact Elevate Software immediately for a resolution to the problem.
DBISAM_LOCKWRITELOCK (10223)	Cannot write lock the lock file for the database '<DatabaseName>'This error occurs when DBISAM internally attempts to place a write lock on the lock file (dbisam.lck) for a given database and it fails. Under normal circumstances this error should never occur so if it does you should contact Elevate Software immediately for a resolution to the problem. Please see the Locking and Concurrency topic for more information.
DBISAM_LOCKWRITEUNLOCK (10224)	Cannot write unlock the lock file for the database '<DatabaseName>'This error occurs when DBISAM internally attempts to remove a write lock on the lock file (dbisam.lck) for a given database and it fails. Under normal circumstances this error should never occur so if it does you should contact Elevate Software immediately for a resolution to the problem.
DBISAM_READLOCK (10225)	Cannot read lock the table '<TableName>'This error occurs when an attempt is made to place a read lock on a table and it fails. Usually this indicates a hardware failure or a failure due to an excessively large amount of write activity on a given table that is preventing the read lock from being acquired. Please see the Locking and Concurrency topic for more information.
DBISAM_READUNLOCK (10226)	Cannot read unlock the table '<TableName>'This error occurs when an attempt is made to remove a read lock on a table and it fails. This error should never occur so if it does you should contact Elevate Software immediately for a resolution to the problem.
DBISAM_WRITELOCK (10227)	Cannot write lock the table '<TableName>'This error occurs when an attempt is made to place a write lock on a table and it fails. Usually this indicates a hardware failure or a failure due to an excessively large amount of read or write activity on a given table that is preventing the write lock from being acquired. Please see the Locking and Concurrency topic for more information.
DBISAM_WRITEUNLOCK (10228)	Cannot write unlock the table '<TableName>'This error occurs when an attempt is made to remove a write lock on a table and it fails. This error should never occur so if it does you should contact Elevate Software immediately for a resolution to the problem.
DBISAM_TRANSLOCK (10229)	Transaction cannot lock the database '<DatabaseName>'This error occurs when an attempt is made to start a transaction and the transaction cannot

	place a special transaction lock on the database. Usually this indicates a hardware failure or a failure due to an excessively large amount of transaction or write activity on the database that is preventing the transaction lock from being acquired. Please see the Locking and Concurrency topic for more information.
DBISAM_TRANSUNLOCK (10230)	Transaction cannot unlock the database '<DatabaseName>'This error occurs when a transaction commits or rolls back and the transaction cannot remove the special transaction lock on the database. This error should never occur so if it does you should contact Elevate Software immediately for a resolution to the problem.
DBISAM_LOCKED (10241)	Cannot lock the table '<TableName>'This error occurs when an attempt is made to lock a table that is already locked by another application or the same application. Since locking a table is the equivalent of locking all of the records in the table (including new records), this error will also occur if an attempt is made to lock a table and there are individual record(s) already locked in the table. Please see the Locking and Concurrency topic for more information.
DBISAM_UNLOCKFAILED (10242)	Cannot unlock the table or record in the table '<TableName>'This error occurs when DBISAM cannot unlock a record in a table or the entire table itself. This error should never occur so if it does you should contact Elevate Software immediately for a resolution to the problem.
DBISAM_NEEDEXCLACCESS (10253)	The table '<TableName>' must be opened exclusivelyThis error occurs when an attempt is made to perform an operation on a table that requires the table to be opened exclusively in order to complete the operation. The operations that require exclusive use of the table are as follows: • Creating and Altering Tables • Verifying and Repairing Tables • Adding and Deleting Indexes from a Table • Optimizing Tables • Upgrading Tables • Emptying Tables
DBISAM_RECLOCKFAILED (10258)	Cannot lock record in the table '<TableName>'This error occurs when an attempt is made to lock a record in a table that is already locked by another application or the same application. Since locking a table is the equivalent of locking all of the records in the table (including new records), this error will also occur if an attempt is made to lock a record and there the table is already locked.

	Please see the Locking and Concurrency topic for more information.
DBISAM_NOTSUFFTABLERIGHTS (10498)	Insufficient rights to the table '<TableName>', a password is requiredThis error occurs when an attempt is made to perform an operation on an encrypted table and a valid password has not been provided for the current session. Please see the Starting Sessions and Opening Tables topics for more information.
DBISAM_NOTABLOB (10753)	Invalid BLOB field '<FieldName>' specified for the table '<TableName>'This error occurs when an attempt is made by DBISAM internally to perform a BLOB operation on a field that is not a BLOB field. This error should never occur so if it does you should contact Elevate Software immediately for a resolution to the problem.
DBISAM_NOTINITIALIZED (10758)	The database engine is not initializedThis error occurs when an attempt is made to use DBISAM without first intializing the database engine. This error should never occur so if it does you should contact Elevate Software immediately for a resolution to the problem.
DBISAM_OSENOENT (11010)	Table or backup file '<Name>' does not existThis error occurs when an attempt is made to open a table or backup file that does not exist. Please see the Opening Tables and Backing Up and Restoring Databases topics for more information.
DBISAM_OSEMFILE (11012)	Too many operating system files open while attempting to open the table or backup file '<Name>'This error occurs when an attempt is made to open up a table or backup file and the operating system rejects the open because it has run out of available file handles. This error should never occur so if it does you should contact Elevate Software immediately for a resolution to the problem.
DBISAM_OSEACCES (11013)	Access denied to table or backup file '<Name>'This error usually occurs when an attempt is made to open up a table or backup file and the operating system does not allow access to the table or backup file due to the fact that it has already been opened up exclusively by another application or the same application or a user rights issue. A table can be opened exclusively only once. It can also occur when an attempt is made to open up a table for read/write access that is in a read-only directory or on a read-only drive and not marked as read-only as an attribute. Please see the Opening Tables topic for more information. It is also possible that a table's backup files cannot be overwritten during the processing of altering the structure of a table, adding indexes to a table or deleting indexes from a table, or optimizing a table. Please see the Creating and Altering Tables, Adding and Deleting Indexes from a Table, and Optimizing Tables topics for more information.
DBISAM_OSEBADF (11014)	Invalid operating system file handle for the table or backup file '<Name>'This error occurs when DBISAM

	internally attempts to perform an operation on a table or backup file using an invalid file handle. This error should never occur so if it does you should contact Elevate Software immediately for a resolution to the problem.
DBISAM_OSENOMEM (11016)	There is insufficient operating system memory available for the current operationThis error is reported by the memory manager in Delphi or C++Builder when there is insufficient memory available for current operation. If you receive this error you should contact Elevate Software in order to find out how to reduce the amount of memory being consumed and to find alternate solutions to the problem.
DBISAM_OSENODEV (11023)	Access denied to logical operating system device for the table or backup file '<Name>'This error occurs when the operating system reports that the logical device for the table or backup file is invalid or inaccessible. This problem can occur with removable disks in floppy drives or CD-ROM drives.
DBISAM_OSENOTSAM (11025)	The table or backup file '<Name>' has been movedThis error occurs when the operating system detects that a table or backup file has been moved from one device to another after being opened. This error should never occur so if it does you should contact Elevate Software immediately for a resolution to the problem.
DBISAM_OSUNKNOWN (11047)	An unknown operating system error <OSErrorCode> occurred with the table or backup file '<Name>'This error occurs when DBISAM detects a general operating system error, indicated by the error code in the error message, but cannot translate it into a more specific error message. DBISAM identifies and translates certain operating system errors such as sharing violations, file and directory not found errors, etc. into specific error messages. If you receive this error you should contact Elevate Software immediately for a resolution to the problem.
DBISAM_REMOTECOMMLOST (11276)	The connection to the database server at '<RemoteName>' has been lostThis error occurs when an application attempts an operation on a database server, the operation fails, and DBISAM is unable to automatically re-connect to the database server. This is most often caused by a physical interruption in the connection between the client and server processes. Please see the Starting Sessions topic for more information.
DBISAM_REMOTEENCRYPTREQ (11277)	The database server at '<RemoteName>' requires an encrypted connectionThis error occurs when an application attempts a regular connection to a database server and the database server requires encrypted connections only. This error can also occur if an administrator is attempting to connect to the administrative port on the database server and the connection is unencrypted. Administrative connections always require encryption. Please see the Starting

	Sessions and Server Administration topics for more information.
DBISAM_REMOTEUNKNOWN (11279)	An unknown error ('<Message>') occurred with the connection to the database server at '<RemoteName>', please check the server logThis error occurs when an unknown error has occurred during an operation on the database server. You should check the server log as soon as possible using an administrative connection in order to find out the nature of the error. Please see the Server Administration topic for more information.
DBISAM_REMOTECONNECT (11280)	A connection to the database server at '<RemoteName>' cannot be establishedThis error occurs when an application attempts to connect to a database server via a TDBISAMSession component and the connection fails. This is most often caused by the lack of a server process listening on the specified port at the specified IP address. Please see the Starting Sessions topic for more information.
DBISAM_REMOTENOLOGIN (11281)	A connection to the database server at '<RemoteName>' cannot be established, the server is not accepting new loginsThis error occurs when an application attempts to connect to a database server and the connection fails because the server administrator has configured the server to refuse any new logins. Please see the Starting Sessions and Server Administration topics for more information.
DBISAM_REMOTEMAXCONNECT (11282)	A connection to the database server at '<RemoteName>' cannot be established, the maximum number of server or user connections has been reachedThis error occurs when an application attempts to connect to a database server and the connection fails because the maximum number of connections configured for the server would be exceeded by the new connection. This error can also occur when a user connects and logs in to the database server and the maximum number of connections configured for that user would be exceeded by the new connection. Please see the Starting Sessions and Server Administration topics for more information.
DBISAM_REMOTEADDRESSBLOCK (11283)	A connection to the database server at '<RemoteName>' cannot be established, the client address is blockedThis error occurs when an application attempts to connect to a database server and the connection fails because the IP address of the machine the application is running on has been blocked in the server configuration. Please see the Starting Sessions and Server Administration topics for more information.
DBISAM_REMOTECALLBACKERR (11285)	A server callback error occurred for the database server at '<RemoteName>'This error occurs when an internal callback used by DBISAM when connected to a database server fails for some reason. This error should never occur so if it does you should contact Elevate Software immediately for a resolution to the problem.

DBISAM_REMOTEVERSION (11286)	A call to the database server at '<RemoteName>' failed, the client engine version does not match the serverThis error occurs when an application attempts to execute functionality that results in a call to a database server, and the call fails because the DBISAM version of the application does not match the required version of the database server for that particular call. Only in certain cases where database server calls have been altered due to an enhancement or bug fix will this error be seen, and those cases will be documented in an incident report. Please see the Starting Sessions topic for more information.
DBISAM_REMOTEINVLOGIN (11287)	A connection to the database server at '<RemoteName>' cannot be established, the login information provided is invalidThis error occurs when an application attempts to connect to a database server and the connection fails because the login information (user name and/or password) specified is invalid. Please see the Starting Sessions topic for more information.
DBISAM_REMOTENOTAUTH (11288)	The user '<UserName>' is not authorized to perform this operation with the database '<DatabaseName>' on the database server at '<RemoteName>'This error occurs when an application attempts to perform a function on a database server and it fails because the user has not been granted rights to perform such a function on the current database. Please see the Server Administration topic for more information.
DBISAM_REMOTENOTADMIN (11289)	The user '<UserName>' is not authorized to perform administration functions on the database server at '<RemoteName>'This error occurs when an application attempts to perform an administrative function on a database server and it fails because the user is not designated as an administrator for that databse server. Please see the Server Administration topic for more information.
DBISAM_REMOTEINVUSER (11290)	The user name is either invalid or blankThis error occurs when an attempt is made to add a new user to a database server and the user name is either missing or invalid. Only administrators can add new users to a database server. Please see the Server Administration topic for more information.
DBISAM_REMOTENOUSER (11291)	The user '<UserName>' does not exist on the database server at '<RemoteName>'This error occurs when an application attempts to edit or remove a user on a database server that does not exist. Only administrators can edit or remove users from a database server. Please see the Server Administration topic for more information.
DBISAM_REMOTEDUPUSER (11292)	The user '<UserName>' already exists on the database server at '<RemoteName>'This error occurs when an attempt is made to add a new user to a database server and the user name (case-insensitive) already exists on that database server. Only administrators can add users to a database server. Please see the Server

	Administration topic for more information.
DBISAM_REMOTEINVDB (11293)	The database name or directory is either invalid or blankThis error occurs when an attempt is made to add a new database to a database server and the database name is either missing or invalid or the directory specified cannot be created from the server. Only administrators can add databases to a database server. Please see the Server Administration topic for more information.
DBISAM_REMOTENODB (11294)	The database '<DatabaseName>' does not exist on the database server at '<RemoteName>'This error occurs when an application attempts to edit or remove a database that does not exists from a database server. Only administrators can edit or remove databases from a database server. Please see the Server Administration topic for more information.
DBISAM_REMOTEDUPDB (11295)	The database '<DatabaseName>' already exists on the database server at '<RemoteName>'This error occurs when an attempt is made to add a new database to a database server and the database name (case-insensitive) already exists on that database server. Only administrators can add databases to a database server. Please see the Server Administration topic for more information.
DBISAM_REMOTEINVDBUSER (11296)	The database user name is either invalid or blankThis error occurs when an attempt is made to add a new user to a database on a database server and the user name is either missing or invalid. Only administrators can add users to a given database on a database server. Please see the Server Administration topic for more information.
DBISAM_REMOTENODBUSER (11297)	The database user '<UserName>' does not exist for the database '<DatabaseName>' on the database server at '<RemoteName>'This error occurs when an application attempts to edit or remove a user that does not exist from a database on a database server. Only administrators can edit or remove users from a given database on a database server. Please see the Server Administration topic for more information.
DBISAM_REMOTEDUPDBUSER (11298)	The database user '<UserName>' already exists for the database '<DatabaseName>' on the database server at '<RemoteName>'This error occurs when an attempt is made to add a new user to a database on a database server and the user name (case-insensitive) already exists for the specified database on that database server. Only administrators can add users to a given database on a database server. Please see the Server Administration topic for more information.
DBISAM_REMOTEINVPROC (11299)	The procedure name is either invalid or blankThis error occurs when an attempt is made to add a new server-side procedure to a database server and the procedure name is either missing or invalid. Only administrators can add server-side procedures to a database server. Please

	see the Server Administration and Customizing the Engine topics for more information.
DBISAM_REMOTENOPROC (11300)	The procedure '<ProcedureName>' does not exist on the database server at '<RemoteName>'This error occurs when an application attempts to edit or remove a server-side procedure that does not exist from a database server. Only administrators can edit or remove server-side procedures from a database server. Please see the Server Administration and Customizing the Engine topics for more information.
DBISAM_REMOTEDUPPROC (11301)	The procedure '<ProcedureName>' already exists on the database server at '<RemoteName>'This error occurs when an attempt is made to add a new server-side procedure to a database server and the procedure name (case-insensitive) already exists on that database server. Only administrators can add server-side procedures to a database server. Please see the Server Administration and Customizing the Engine topics for more information.
DBISAM_REMOTEINVPROCUSER (11302)	The procedure user name is either invalid or blankThis error occurs when an attempt is made to add a new user to a server-side procedure on a database server and the user name is either missing or invalid. Only administrators can add users to a given server-side procedure on a database server. Please see the Server Administration and Customizing the Engine topics for more information.
DBISAM_REMOTENOPROCUSER (11303)	The procedure user '<UserName>' does not exist for the procedure '<ProcedureName>' on the database server at '<RemoteName>'This error occurs when an application attempts to edit or remove a user that does not exist from a server-side procedure on a database server. Only administrators can edit or remove users from a given server-side procedure on a database server. Please see the Server Administration and Customizing the Engine topics for more information.
DBISAM_REMOTEDUPPROCUSER (11304)	The procedure user '<UserName>' already exists for the procedure '<ProcedureName>' on the database server at '<RemoteName>'This error occurs when an attempt is made to add a new user to a server-side procedure on a database server and the user name (case-insensitive) already exists for the specified procedure on that database server. Only administrators can add users to a given server-side procedure on a database server. Please see the Server Administration and Customizing the Engine topics for more information.
DBISAM_REMOTEINVEVENT (11305)	The event name is either invalid or blankThis error occurs when an attempt is made to add a new scheduled event to a database server and the event name is either missing or invalid. Only administrators can add scheduled events to a database server. Please see the Server Administration and Customizing the Engine topics for more information.

DBISAM_REMOTENOEVENT (11306)	The event '<EventName>' does not exist on the database server at '<RemoteName>'This error occurs when an application attempts to edit or remove a scheduled event that does not exist from a database server. Only administrators can edit or remove scheduled events from a database server. Please see the Server Administration and Customizing the Engine topics for more information.
DBISAM_REMOTEDUPEVENT (11307)	The event '<EventName>' already exists on the database server at '<RemoteName>'This error occurs when an attempt is made to add a new scheduled event to a database server and the event name (case-insensitive) already exists on that database server. Only administrators can add scheduled events to a database server. Please see the Server Administration and Customizing the Engine topics for more information.
DBISAM_REMOTEINVREQUEST (11308)	An invalid or unknown request was made to the database server at '<RemoteName>'This error occurs when an application makes a call to a database server that is invalid or malformed. This error should never occur under normal operation, but if it does it usually indicates an improper client engine or an attempt to break into the server.
DBISAM_EXPORTERROR (11310)	An error occurred during the export from the table '<TableName>' - <Message>This error occurs when any error is encountered during the export of a table to a text file. The message will give the specific details of the error and why the error occurred. Please see the Importing and Exporting Tables topic for more information.
DBISAM_IMPORTERROR (11312)	An error occurred during the import into the table '<TableName>' - <Message>This error occurs when any error is encountered during the import of a text file into a table. The message will give the specific details of the error and why the error occurred. Please see the Importing and Exporting Tables topic for more information.
DBISAM_LOADSTREAMERROR (11312)	An error occurred during the loading of a stream into the table '<TableName>' - <Message>This error occurs when any error is encountered during the loading of a stream into a table. The message will give the specific details of the error and why the error occurred. Please see the Loading and Saving Streams topic for more information.
DBISAM_SAVESTREAMERROR (11313)	An error occurred during the saving of the table '<TableName>' to a stream - <Message>This error occurs when any error is encountered during the saving of a table or query to a stream. The message will give the specific details of the error and why the error occurred. Please see the Loading and Saving Streams topic for more information.
DBISAM_TRIGGERERROR (11314)	An error occurred during a trigger on the table

	'<TableName>' - <Message>This error occurs when any error is encountered during the execution of an insert, update, or delete trigger for a table. The message will give the specific details of the error and why the error occurred. Please see the Customizing the Engine topic for more information.
DBISAM_SQLPARSE (11949)	SQL parsing error - <Message>This error occurs when any error is encountered during the parsing or preparation of an SQL statement. The message will give the specific details of the error and why the error occurred. Please see the TDBISAMQuery OnQueryError event for more information on trapping these kind of errors.
DBISAM_SQLEXEC (11950)	SQL execution error - <Message>This error occurs when any error is encountered during the execution of an SQL statement. The message will give the specific details of the error and why the error occurred. Please see the TDBISAMQuery OnQueryError event for more information on trapping these kind of errors.
DBISAM_OLDVERSION (12035)	The table '<TableName>' is not the correct versionThis error occurs when an attempt is made to open a table that is either the incorrect version for the current version of DBISAM or is not a DBISAM table at all (but may have the same extensions as the physical files that make up a logical DBISAM table). Please see the Upgrading Tables topic for more information.
DBISAM_BADSIGNATURE (12036)	The table or backup file '<Name>' is not validThis error occurs when an attempt is made to open a table or backup file that was created using a different engine signature than the current engine signature in use. It is also possible that the file is not a DBISAM table at all (but may have the same extensions as the physical files that make up a logical DBISAM table).
DBISAM_SEARCHCOLREQD (12292)	Invalid field type specified for the index '<IndexName>' in the table '<TableName>'This error occurs when an attempt is made to add a primary or secondary index, create a table, or alter an existing table's structure and one or more of the indexes contain BLOB fields as one of the index fields. BLOB fields cannot be directly indexed in DBISAM and must be indexed using the full text indexing instead. Please see the Full Text Indexing topic for more information.
DBISAM_TABLEEXISTS (13060)	The table '<TableName>' already existsThis error occurs when an attempt is made to create a table and the table already exists. Please see the Creating and Altering Tables topic for more information.
DBISAM_COMPRESS (15001)	Error compressing dataThis error occurs when DBISAM attempts to compress a buffer and the compression fails. This error should never occur, so if you receive this error you should immediately contact Elevate Software for more information on how to resolve this issue.

DBISAM_UNCOMPRESS (15002)	Error uncompressing dataThis error occurs when DBISAM attempts to uncompress a buffer and the uncompression fails. This error should never occur, so if you receive this error you should immediately contact Elevate Software for more information on how to resolve this issue.
DBISAM_CANNOTLOADLDDRV (15878)	The locale support for the table '<TableName>' is not available or installedThis error occurs when an attempt is made to open or create a table with a locale that is not available or installed on the current machine's operating system. Please see the Creating and Altering Tables topic for more information.

Appendix C - System Capacities

The following is a detailed list of the capacities for the different components in DBISAM. Unless where noted all capacities are absolute and not dependent upon the underlying operating system or network.

Capacity	Details
# of BLOB Fields in a Table	The maximum number of BLOB fields in a table is 128.
# of Decimal Places in a BCD Field	The maximum number of decimals places in a BCD field is 4.
# of Fields in Table	The maximum number of fields in a table is 1024.
# of Fields in an Index Key	The maximum number of fields in an index key is 128.
# of Indexes in a Table	The maximum number of primary indexes in a table is 1. The maximum number of secondary indexes in a table is 30.
# of Open BLOBs in a Table	The maximum number of open BLOBs in a table is 128.
# of Open Tables	The maximum number of open tables is only limited by the available memory constraints of the operating system or hardware. **Note** DBISAM has a limit of 4096 tables per database, but a single application can open the same table in the same database many times as well as open many tables from other databases.
# of Records in a Table	The maximum number of records in a table is 1 billion.
# of Records in a Transaction	The maximum number of records in a single transaction is only limited by the available memory constraints of the operating system or hardware.
# of Tables in a Database	The maximum number of tables in a database is 4096 tables.
Size Of Database Names	The maximum size of a database name is 60 bytes.
Size Of Field Constraint	The maximum size of a field constraint expression is 100 bytes.
Size Of Field Descriptions	The maximum size of a field description is 100 bytes.
Size Of Field Names	The maximum size of a field name is 60 bytes.
Size Of Index Names	The maximum size of an index name is 60 bytes.
Size Of Physical Table Files	The maximum size of the physical files that make up a table - <TableName>.dat for data, <TableName>.idx for indexes, and <TableName>.blb for BLOBS, is 128,000,000,000 bytes for each file.
Size Of Table Descriptions	The maximum size of a table description is 100 bytes.

Size Of Table Names	The maximum size of a table name is 60 bytes.
Size of BLOB Blocks	The minimum BLOB block size is 64 bytes and the maximum BLOB block size is 64 kilobytes.
Size of BLOB Fields	The maximum length of a BLOB field is 2 gigabytes.
Size of In-Memory Tables	The maximum length of an in-memory table is only limited by the available memory constraints of the operating system or hardware.
Size of Index Keys	The maximum size of an index key is 4096 bytes. The index key length is not an exact sum of the length of the fields present in the index key. The proper way to calculate the index key length is as follows: For primary indexes the calculation is: ((Sum of field sizes) + Number of Fields in Index Key For secondary indexes the calculation is: ((Sum of field sizes) + Number of Fields in Index Key + 4 The extra bytes for the number of fields are the NULL flags used to order the index keys properly based upon whether they are NULL or not. The additional four bytes for secondary indexes is used to store a unique record ID which is essential for proper bookmark support. **Note** The length of string fields in index keys includes the null terminator character. For example, if a string field has a length of 15 then its length in an index key would be 16.
Size of Index Pages	The minimum index page size is 1024 bytes and the maximum index page size is 16 kilobytes.
Size of Records	The maximum record size is 65280 bytes.
Size of String Fields	The maximum length of a string field is 512 bytes. This figure does not include a NULL terminator character. The NULL terminator character is handled internally.